Zero to Monero: Second Edition

a technical guide to a private digital currency; for beginners, amateurs, and experts

Published April 4, 2020 (v2.0.0)

koe[1], Kurt M. Alonso[2], Sarang Noether[3]

[1] ukoe@protonmail.com
[2] kurt@oktav.se
[3] sarang.noether@protonmail.com

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.

It is common knowledge that 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 assign and transfer ownership of funds. To ensure that no piece of money can be duplicated or created at will, cryptocurrencies usually rely on 'blockchains', which are public, distributed ledgers containing records of currency transactions that can be verified by third parties [97].

It might seem at first glance that transactions need to be sent and stored in plain text format to make them publicly verifiable. In fact, it is possible to conceal a transaction's participants, as well as the amounts involved, using cryptographic tools that nevertheless allow transactions to be verified and agreed upon by observers [136]. 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 [97] where transactions are based on elliptic curve cryptography using curve Ed25519 [38], transaction inputs are signed with Schnorr-style multilayered linkable spontaneous anonymous group signatures (MLSAG) [108], and output amounts (communicated to recipients via ECDH [52]) are concealed with Pedersen commitments [88] and proven in a legitimate range with Bulletproofs [43]. Much of the first part of this report is spent explaining these ideas.

Contents

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 trust 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 cryptographic 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. 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 or fungibility[2], since it literally *publicizes* the complete transaction histories of its users.

[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.

[2] "**Fungible** means capable of mutual substitution in use or satisfaction of a contract. ... Examples: ... money, etc."[25] In an open blockchain such as Bitcoin, the coins owned by Alice can be differentiated from those owned by Bob based on the 'transaction history' of those coins. If Alice's transaction history includes transactions related to supposedly nefarious actors, then her coins might be 'tainted' [95], and hence less valuable than Bob's (even if they own the same amount of coins). Reputable figures claim that newly minted Bitcoins trade at a premium over used coins, since they don't have a history [118].

To address the lack of privacy, users of cryptocurrencies such as Bitcoin can obfuscate transactions by using temporary intermediate addresses [99]. However, with appropriate tools it is possible to analyze flows and to a large extent link true senders with receivers [125, 41, 112, 48].

In contrast, the cryptocurrency Monero (Moe-neh-row) attempts to tackle the issue of privacy by storing only 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 known generally effective ways to link receivers or trace the origin of funds.[3]

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 and fungibility.

1.1 Objectives

Monero is an established cryptocurrency with over five years of development [127, 13], and maintains a steadily increasing level of adoption [54].[4] Unfortunately, there is little comprehensive documentation describing the mechanisms it uses.[5,6] Even worse, essential parts of its theoretical framework have been published in non-peer-reviewed papers that are incomplete or contain errors. For significant parts of the theoretical framework of Monero, only the source code is reliable as a source of information.[7]

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.[8]

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.

[3] Depending on the behavior of users, there may be cases where transactions can be analyzed to some extent. For an example see this article: [61].

[4] In terms of market capitalization, Monero has been steady relative to other cryptocurrencies. It was 14th as of June 14th, 2018, and 12th on January 5th, 2020; see `https://coinmarketcap.com/`.

[5] One documentation effort, at `https://monerodocs.org/`, has some helpful entries, especially related to the Command Line Interface. The CLI is a Monero wallet accessible through a console/terminal. It has the most functionality out of all Monero wallets, at the expense of no user-friendly graphical interface.

[6] Another, more general, documentation effort called Mastering Monero can be found here: [57].

[7] Mr. Seguias has created the excellent Monero Building Blocks series [123], which contains a thorough treatment of the cryptographic security proofs used to justify Monero's signature schemes. As with Zero to Monero: First Edition [33], Seguias's series is focused on v7 of the protocol.

[8] A previous attempt to explain how Monero works [110] did not elucidate elliptic curve cryptography, was incomplete, and is now over five years outdated.

In the second edition of this report we have centered our attention on version 12 of the Monero protocol[9], corresponding to version 0.15.x.x of the Monero software suite. All transaction and blockchain-related mechanisms described here belong to those versions.[10,11] Deprecated transaction schemes have not been explored to any extent, even if they may be partially supported for backward compatibility. Likewise with deprecated blockchain features. The first edition [33] corresponded to version 7 of the protocol, and version 0.12.x.x of the software suite.

1.2 Readership

We anticipate many readers will encounter this report with little to no understanding of discrete mathematics, algebraic structures, cryptography[12], and blockchains. We have tried to be thorough enough that laypeople from all perspectives may learn Monero without needing external research.

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.[13]

1.3 Origins of the Monero cryptocurrency

The cryptocurrency Monero, initially known as BitMonero, was created in April 2014 as a derivative of the proof-of-concept currency CryptoNote [127]. 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 [136]. It offered receiver anonymity through the use of one-time addresses, and sender ambiguity by means of ring signatures.

[9] The 'protocol' is the set of rules that each new block is tested against before it can be added to the blockchain. This set of rules includes the 'transaction protocol' (currently version 2, RingCT), which are general rules pertaining to how a transaction is constructed. Specific transaction rules can, and do, change, without the transaction protocol's version changing. Only large-scale changes to the transaction structure warrant moving its version number.

[10] The Monero codebase's integrity and reliability is predicated on assuming enough people have reviewed it to catch most or all significant errors. We hope that readers will not take our explanations for granted, and verify for themselves the code does what it's supposed to. If it doesn't, we hope you will make a responsible disclosure (`https://hackerone.com/monero`) for major problems, or Github pull request (`https://github.com/monero-project/monero`) for minor issues.

[11] Several protocols worth considering for the next generation of Monero transactions are undergoing research and investigation, including Triptych [106], RingCT3.0 [143], Omniring [84], and Lelantus [71].

[12] An extensive textbook on applied cryptography can be found here: [42].

[13] Some footnotes, especially in chapters related to the protocol, spoil future chapters or sections. These are intended to make more sense on a second read-through, since they usually involve specific implementation details that are only useful to those who have a grasp of how Monero works.

Since its inception, Monero has further strengthened its privacy aspects by implementing amount hiding, as described by Greg Maxwell (among others) in [89] and integrated into ring signatures based on Shen Noether's recommendations in [108], then made more efficient with Bulletproofs [43].

1.4 Outline

As mentioned, our aim is to deliver a self-contained and step-by-step description of the Monero cryptocurrency. Zero to Monero has been structured to fulfill this objective, leading the reader through all parts of the currency's inner workings.

1.4.1 Part 1: 'Essentials'

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 expands on the Schnorr signature scheme from the prior chapter, and outlines the ring signature algorithms that will be applied to achieve confidential transactions.

Chapter 4 explores how Monero uses addresses to control ownership of funds, and the different kinds of addresses.

In Chapter 5 we introduce the cryptographic mechanisms used to conceal amounts.

With all the components in place, we explain the transaction scheme used in Monero in Chapter 6.

The Monero blockchain is unfolded in Chapter 7.

1.4.2 Part 2: 'Extensions'

A cryptocurrency is more than just its protocol, and in 'Extensions' we talk about a number of different ideas, many of which have not been implemented.[14]

Various information about a transaction can be proven to observers, and those methods are the content of Chapter 8.

While not essential to the operation of Monero, there is a lot of utility in multisignatures that allow multiple people to send and receive money collaboratively. Chapter 9 describes Monero's current multisignature approach and outlines possible future developments in that area.

[14] Please note that future protocol versions of Monero, especially those implementing new transaction protocols, may make any or all of these ideas impossible or impractical.

Of extreme importance is applying multisig to the interactions of vendors and shoppers in online marketplaces. Chapter 10 constitutes our original design of an escrowed marketplace using Monero multisig.

First presented here, TxTangle, outlined in Chapter 11, is a decentralized protocol for joining the transactions of multiple individuals into one.

1.4.3 Additional content

Appendix A explains the structure of a sample transaction from the blockchain. Appendix B explains the structure of blocks (including block headers and miner transactions) in Monero's blockchain. Finally, Appendix C 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.[15] There is usually a file path, such as src/ringct/rctOps.cpp, and a function, such as `ecdhEncode()`. Note: '-' indicates split text, such as crypto- note → cryptonote, and we neglect namespace qualifiers (e.g. `Blockchain::`) in most cases.

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 primary 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. In particular, as a common saying goes, 'don't roll your own crypto'. Code implementing cryptographic primitives should be well-reviewed by experts and have a long history of dependable performance. Moreover, original contributions in this document may not be well-reviewed and are likely untested, so readers should exercise their judgement when reading them.

1.6 History of Zero to Monero

Zero to Monero is an expansion of Kurt Alonso's master's thesis, 'Monero - Privacy in the Blockchain' [32], published in May 2018. The first edition was published in June 2018 [33].

For the second edition we have improved how ring signatures are introduced (Chapter 3), reorganized how transactions are explained (added Chapter 4 on Monero Addresses), modernized the

[15] Our margin notes are accurate for version 0.15.x.x of the Monero software suite, but may gradually become inaccurate as the codebase is constantly changing. However, the code is stored in a git repository (https://github.com/monero-project/monero), so a complete history of changes is available.

method used to communicate output amounts (Section 5.3), replaced Borromean ring signatures with Bulletproofs (Section 5.5), deprecated `RCTTypeFull` (Chapter 6), updated and elaborated Monero's dynamic block weight and fee system (Chapter 7), investigated transaction-related proofs (Chapter 8), described Monero multisignatures (Chapter 9), designed solutions for escrowed marketplaces (Chapter 10), proposed a new decentralized joint transaction protocol named TxTangle (Chapter 11), updated or added various minor details to align with the most current protocol (v12) and Monero software suite (v0.15.x.x), and scoured the document for quality-of-reading edits.[16]

1.7 Acknowledgements

Written by author 'koe'.

This report would not exist without Kurt's original master's thesis [32], so to him I owe a great debt of gratitude. Monero Research Lab (MRL) researchers Brandon "Surae Noether" Goodell and pseudonymous 'Sarang Noether' (who collaborated with me on Section 5.5 and Chapter 8) have been a reliable and knowledgeable resource throughout both editions of Zero to Monero's development. Pseudonymous 'moneromooo', the Monero Project's most prolific core developer, has probably the most extensive understanding of the codebase on this planet, and has pointed me in the right direction numerous times. And of course, many other wonderful Monero contributors have spent time answering my endless questions. Finally, to the various individuals who contacted us with proofreading feedback, and encouraging comments, thank you!

[16] The LaTeX source code for both editions of Zero to Monero can be found here (the first edition is in branch 'ztm1'): https://github.com/UkoeHB/Monero-RCT-report.

Part I

Essentials

CHAPTER 2

Basic Concepts

2.1 A few words about notation

A 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.[1]

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

src/crypto/keccak.c

A cryptographic hash function (henceforth simply 'hash function', or 'hash') 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 similar to the 'remainder' after dividing two numbers, e.g. c the 'remainder' of a/b. Let's imagine a number line. To calculate $c = a \pmod b$ 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 (modulo 3) $= 1$, $-5 \pmod 4 = 3$, and so on.

[1] In computer memory, each byte is stored in its own address (an address is akin to a numbered slot, which a byte can be stored in). A given 'word' or variable is referenced by the lowest address of its bytes. If variable x has 4 bytes, stored in addresses 10-13, address 10 is used to find x. The way bytes of x are organized in its set of addresses depends on *endianness*, although each individual byte is always and everywhere stored the same way within its address. Basically, which end of x is stored in the reference address? It could be the *big end* or *little end*. Given $x =$ 0x12345678 (hexadecimal, 2 hexadecimal digits occupy 1 byte e.g. 8 binary digits a.k.a. bits), and an array of addresses {10, 11, 12, 13}, the big endian encoding of x is {12, 34, 56, 78} and the little endian encoding is {78, 56, 34, 12}. [78]

[2] The Keccak hashing algorithm forms the basis for the NIST standard *SHA-3* [27].

Formally, the positive modulus is here defined for $c = a \pmod b$ as $a = bx + c$, where $0 \leq c < b$ and x is a signed integer which gets discarded (b is a positive non-zero integer).

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

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 ($a * b \pmod n = c$) with an algorithm called 'double-and-add'. Let us demonstrate by example. Say we want to do $7 * 8 \pmod 9 = 2$. It 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.[3]

Double-and-add is implemented by converting the first number (the 'multiplicand' a) 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 3,2,1,0.[4] A[0] = 1 is the first element of A and is the least significant bit. We set a result variable to be initially $r = 0$, and set a sum variable to be initially $s = 8$ (more generally, we start with $s = b$). We follow this algorithm:

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

[3] The effect of double-and-add becomes apparent with large numbers. For example, with $2^{15} * 2^{30}$ straight addition would require about 2^{15} + operations, while double-and-add only requires 15!

[4] This is known as 'LSB 0' numbering, since the least significant bit has index 0. We will use 'LSB 0' for the rest of this chapter. The point here is clarity, not accurate conventions.

In our example $7 * 8 \pmod 9$, this sequence appears:

1. $i = 0$
   (a) A[0] = 1, so $r = 0 + 8 \pmod 9 = 8$
   (b) $s = 8 + 8 \pmod 9 = 7$
2. $i = 1$
   (a) A[1] = 1, so $r = 8 + 7 \pmod 9 = 6$
   (b) $s = 7 + 7 \pmod 9 = 5$
3. $i = 2$
   (a) A[2] = 1, so $r = 6 + 5 \pmod 9 = 2$
   (b) $s = 5 + 5 \pmod 9 = 1$
4. $i = 3$
   (a) A[3] = 0, so r stays the same
   (b) $s = 1 + 1 \pmod 9 = 2$
5. $r = 2$ is the result

2.2.2 Modular exponentiation

Clearly $8^7 \pmod 9 = 8 * 8 * 8 * 8 * 8 * 8 * 8 \pmod 9$. 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: $i = (0, ..., A_{size} - 1)$
   (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 $1/a \pmod n$, or in other words $a^{-1} \pmod n$. The inverse of something times itself is by definition one (identity). Imagine $0.25 = 1/4$, and then $0.25 * 4 = 1$.

In modular arithmetic, for $c = a^{-1} \pmod n$, $ac \equiv 1 \pmod n$ for $0 \leq c < n$ and for a and n relatively prime.[5] Relatively prime means they don't share any divisors except 1 (the fraction a/n can't be reduced/simplified).

We can use square-and-multiply to compute the modular multiplicative inverse when n is a prime number because of *Fermat's little theorem*:[6]

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

More generally (and more rapidly), the so-called 'extended Euclidean algorithm' [7] can also find modular inverses.

2.2.4 Modular equations

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

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

Now we have a way to do modular subtraction.

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

The same principle would apply to something like $x = (a - b * c * d)^{-1}(e * f + g^h) \pmod n$.[7]

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

[6] 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]
It means we can do $c = a^{-1}b \pmod n \rightarrow ca \equiv b \pmod n \rightarrow a \equiv c^{-1}b \pmod n$.

[7] The modulus of large numbers can exploit modular equations. It turns out $254 \pmod{13} \equiv 2{*}10{*}10{+}5{*}10{+}4 \equiv (((2) * 10 + 5) * 10 + 4) \pmod{13}$. An algorithm for $a \pmod n$ when $a > n$ is:

1. Define $A \rightarrow [a_{decimal}]$; $r = 0$
2. For $i = A_{size} - 1, ..., 0$
   (a) $r = (r * 10 + A[i]) \pmod n$
3. Use the final r as result.

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 $(\text{mod } q)$.

`fe`: field element

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

Typically, an elliptic curve with a given (a, b) pair is defined as the set of all points with coordinates (x, y) satisfying a *Weierstraß* equation [69]:[8]

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

The cryptocurrency Monero uses a special curve belonging to the category of so-called *Twisted Edwards* curves [38], 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 [40] 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[9]

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

These formulas for addition also apply for point doubling; that is, when $P_1 = P_2$. To subtract a point, invert its coordinates over the y-axis, $(x, y) \to (-x, y)$ [38], and use point addition. Recall that 'negative' elements $-x$ of $\mathbb{F}_q$ are really $-x \pmod{q}$.

Whenever two curve points are added together P_3 is a point on the 'original' elliptic curve, or in other words all $x_3, y_3 \in \mathbb{F}_q$ and satisfy the EC equation.

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

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

[9] Typically elliptic curve points are converted into projective coordinates prior to curve operations like point addition, in order to avoid performing field inversions for efficiency. [142]

src/crypto/crypto_ops_builder/ref10CommentedCombined/ge.h

contain the points $(0, P, 2P, 3P, 4P)$, 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 $(0, 1)$.[10]

Conveniently, $5P + P = P$. This means the subgroup is *cyclic*.[11] 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. In our example, take multiples of point $2P$:

$$2P, 4P, 6P, 8P, 10P \rightarrow 2P, 4P, 1P, 3P, 0$$

Another example: a subgroup with order 6 $(0, P, 2P, 3P, 4P, 5P)$. Multiples of point $2P$:

$$2P, 4P, 6P, 8P, 10P, 12P \rightarrow 2P, 4P, 0, 2P, 4P, 0$$

Here $2P$ has order 3. Since 6 is not prime, not all of its member points recreate the original 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*). We can imagine a set of all EC points $\{0, P_1, ..., P_{N-1}\}$. If N isn't prime, some points will make subgroups with orders equal to divisors of N.

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 (such as 160 bits) prime number and h is the so-called *cofactor* which could be as small as 1 or 2.[12] One point in the subgroup of size l is usually selected to be the generator G as a convention. For every other point P in that subgroup there exists an integer $0 < n \leq l$ satisfying $P = nG$.

`ge`: group element

Let's expand our understanding. 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. Since $N = hl$, there are only l multiples of h, implying only one subgroup of size l is possible.

[10] It turns out elliptic curves have *abelian group* structure under the addition operation described, since their point-at-infinity's are identity elements. A concise definition of this notion can be found under https://brilliant.org/wiki/abelian-group/.

[11] Cyclic subgroup means, for P's subgroup with order u, and with any integer n, $nP = [n \pmod u]P$. We can imagine ourselves standing on a point on a globe some distance from a 'zero'th position, and each step we take moves us that distance. After a while, we will wind up back where we started, although it may take many revolutions before we land exactly on the original spot again. The number of steps it takes to land on the exact same spot is the 'order' of the 'stepping group', and all our footprints are unique points in that group. We recommend applying this concept to other ideas discussed here.

[12] EC with small cofactors allow relatively faster point addition, etc. [38].

Put simply, 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 such variations of n' (mod N) (including the point at infinity for $n' = hl$) because when $n' = hl$ it cycles back to 0: $hlP' = 0$. So, there are only h points P in EC where hP will equal 0.

A similar argument could be applied to any subgroup of size u. Any two points P_1 and P_2 with order u are in the same subgroup, which is composed of multiples of $(N/u)P'$.

With this new understanding it is clear we can use the following algorithm to find (non-point-at-infinity) 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.
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 thought to be computationally hard. By analogy to modular arithmetic, this is often called the *discrete logarithm problem* (DLP). Scalar multiplication can be seen as a *one-way function*, which paves the way for using elliptic curves for cryptography.[13]

sc: scalar

The scalar product nP is equivalent to $(((P + P) + (P + P))\dots)$. 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: $i = (0, ..., A_{size} - 1)$
   (a) If A[i] == 1, then R += S.
   (b) Compute S += S.
3. Use the final R as result.

Note that EC scalars for points in the subgroup of size l (which we will be using henceforth) are members of the finite field $\mathbb{F}_l$. This means arithmetic between scalars is mod l.

[13] No known equation or algorithm can efficiently (based on available technology) solve for n in $P_1 = nP_2$, meaning it would take many many years to unravel just one scalar product.

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 $0 < k < l$, and call it a *private key*.[14] Calculate the corresponding *public key* K (an EC point) with the scalar product $kG = K$.

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

2.3.3 Diffie-Hellman key exchange with elliptic curves

A basic *Diffie-Hellman* [52] 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.

   For example, if Alice has a message m to send Bob, she could hash the shared secret $h = \mathcal{H}(S)$, compute $x = m + h$, and send x to Bob. Bob computes $h' = \mathcal{H}(S)$, calculates $m = x - h'$, and learns m.

An external observer would not be able to easily calculate the shared secret due to the 'Diffie-Hellman Problem' (DHP), which says finding S from K_A and K_B is very difficult. Also, the DLP prevents them from finding k_A or k_B.[15]

2.3.4 Schnorr signatures and the Fiat-Shamir transform

In 1989 Claus-Peter Schnorr published a now-famous interactive authentication protocol [121], generalized by Maurer in 2009 [87], that allowed someone to prove they know the private key k of a given public key K without revealing any information about it [89]. It goes something like this:

1. The prover generates a random integer $\alpha \in_R \mathbb{Z}_l$,[16] computes αG, and sends αG to the verifier.

[14] The private key is sometimes known as a *secret key*. This lets us abbreviate: pk = public key, sk = secret key.

[15] The DHP is thought to be of at least similar difficulty to the DLP, although it has not been proven. [31]

[16] Notation: The R in $\alpha \in_R \mathbb{Z}_l$ means α is randomly selected from $\{1, 2, 3, ..., l-1\}$. In other words, $\mathbb{Z}_l$ is all integers (mod l). We exclude 'l' since the point-at-infinity is not useful here.

src/crypto/crypto.cpp `random32_unbiased()`

2. The verifier generates a random *challenge* $c \in_R \mathbb{Z}_l$ and sends c to the prover.
3. The prover computes the *response* $r = \alpha + c * k$ 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 respond with 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 prover then r is randomly distributed [122] and k is information-theoretically secure within r (it can still be found by solving the DLP for K or αG).[17] 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).[18]

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

If the prover knew c from the beginning (e.g. if the verifier secretly gave it to her), she could generate a random response r and compute $\alpha G = rG - cK$. When she later sends r to the verifier, she 'proves' knowledge of k without ever having to know it. Someone observing the transcript of events between prover and verifier would be none the wiser. The scheme is not *publicly verifiable*. [89]

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 [89].[19]

Using a hash function, instead of the verifier, to generate challenges is known as a *Fiat-Shamir transform* [60], because it makes an interactive proof non-interactive and publicly verifiable [89].[20,21]

Non-interactive proof

1. Generate random number $\alpha \in_R \mathbb{Z}_l$, and compute αG.

[17] 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.

[18] 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.

[19] 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." [3]

[20] The output of a cryptographic hash function $\mathcal{H}$ is uniformly distributed across the range of possible outputs. That is to say, 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.

[21] Note that non-interactive Schnorr-like proofs (and signatures) require either use of a fixed generator G, or inclusion of the generator in the challenge hash. Including it that way is known as key prefixing, which we discuss a bit more later (Sections 3.4 and 9.2.3).

2. Calculate the challenge using a cryptographically secure hash function, $c = \mathcal{H}([\alpha G])$.
3. Define the response $r = \alpha + c * k$.
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, keep in mind that verification time is mostly a function of elliptic curve operations.

src/ringct/rctOps.cpp

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 that lets a message's recipient be confident its content is as intended by the signer. One common signature scheme is called ECDSA. See [72], ANSI X9.62, and [69].

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 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 the response r such that $\alpha = r + c * k_A$. In other words, $r = \alpha - c * k_A$.
4. Publish the signature (c, r).

Verification

Any third party who knows the EC domain parameters (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, and need one public EC key.

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 else, a forger without k_A, could have made (c, r) is negligible, so a verifier can be confident the message was not tampered with.

2.4 Curve Ed25519

Monero uses a particular Twisted Edwards elliptic curve for cryptographic operations, *Ed25519*, the *birational equivalent*[22] of the Montgomery curve *Curve25519*.

Both Curve25519 and Ed25519 were released by Bernstein *et al.* [38, 39, 40].[23]

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: src/crypto/crypto_ops_builder/ref10CommentedCombined/ge.h

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

This curve addresses many concerns raised by the cryptography community.[24] It is well known that NIST[25] standard algorithms have issues. For example, it has recently become clear the NIST standard random number generation algorithm PNRG (the version based on elliptic curves) is flawed and contains a potential backdoor [68]. Seen from a broader perspective, standardization authorities like NIST lead to a cryptographic monoculture, introducing a point of centralization. A great example of this was illustrated when the NSA used its influence over NIST to weaken an international cryptographic standard [18].

Curve Ed25519 is not subject to any patents (see [79] for a discussion on this subject), and the team behind it has developed and adapted basic cryptographic algorithms with efficiency in mind [40]. src/crypto/crypto_ops_builder/

Twisted Edwards curves have order expressible 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 digit number (l is 253 bits):[26]

$$2^3 \cdot 7237005577332262213973186563042994240857116359379907606001950938285454250989$$ src/ringct/rctOps.h `curveOrder()`

2.4.1 Binary representation

Elements of $\mathbb{F}_{2^{255}-19}$ are encoded as 256-bit integers, so 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.

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

[23] Dr. Bernstein also developed an encryption scheme known as ChaCha [37, 100], which the primary Monero implementation uses to encrypt certain sensitive information related to users' wallets. src/wallet/ringdb.cpp

[24] Even if a curve appears to have no cryptographic security problems, it's possible the person/organization that created it knows a secret issue that only crops up in very rare curves. Such a person may have to randomly generate many curves in order to find one with a hidden weakness and no known weaknesses. If reasonable explanations are required for curve parameters, then it becomes even more difficult to find weak curves that will be accepted by the cryptographic community. Curve Ed25519 is known as a 'fully rigid' curve, which means its generation process was fully explained. [120]

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

[26] This means private EC keys in Ed25519 are 253 bits.

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 [74]. Point compression for the Ed25519 curve was first described in [39], while the concept was first introduced in [94].

This point compression scheme follows from a transformation of the Twisted Edwards curve equation (assuming $a = -1$, which is true for Monero): $x^2 = (y^2 - 1)/(dy^2 + 1)$,[27] 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: $3 \pmod 5 = 3$, $-3 \pmod 5 = 2$. 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 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.

src/crypto/crypto_ops_builder/ref10CommentedCombined/ge_tobytes.c

Decoding

1. Retrieve the compressed point y', then copy its most significant bit to the parity bit b before setting it to 0. Now it is the original y again.
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[28] $z = uv^3(uv^7)^{(q-5)/8} \pmod q$.
   (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).

src/crypto/crypto_ops_builder/ref10CommentedCombined/ge_frombytes.c

Implementations of Ed25519 (such as Monero) typically use the generator $G = (x, 4/5)$ [39], where x is the 'even', or $b = 0$, variant based on point decompression of $y = 4/5 \pmod q$.

[27] Here $d = -\frac{121665}{121666}$.
[28] 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.[29] 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 [39]. The algorithm can also be found described in Internet RFC8032 [75]. 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* [39].

We provide here an outline of the steps performed by the algorithm. A complete description and sample implementation in the Python language can be found in [75].

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 [75].
2. Calculate αG and the challenge $ch = \mathcal{H}([\alpha G], K, \mathfrak{m})$.
3. Calculate the response $r = \alpha + ch \cdot k$.
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 $2^c rG \stackrel{?}{=} 2^c \alpha G + 2^c ch' * K$ holds then the signature is valid.

The 2^c term comes from Bernstein *et al.*'s general form of the EdDSA algorithm [39]. According to that paper, though it isn't required for adequate verification, removing 2^c provides stronger equations.

[29] See [40] for efficient group operations in Twisted Edwards EC (i.e. point addition, doubling, mixed addition, etc). See [36] for efficient modular arithmetic.

src/crypto/crypto_ops_builder/ref10CommentedCombined/

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, 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 latter method is important in Monero (Section 3.4).

In this signature scheme we store one EC point and one scalar, and have one public EC key.

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 for the EC point αG and scalar r. However, RFC8032 assumes point αG is compressed, which reduces space requirements to only $32 + 32$ bytes. We include the public key K, which implies $32 + 32 + 32$ total bytes.

2.5 Binary operator XOR

The binary operator XOR is a useful tool that will appear in Sections 4.4 and 5.3. It takes two arguments and returns true if one, but not both, of them is true [22]. 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:

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

Each of these also produce $\{0,1\}$: $\text{XOR}(\{1,0\},\{1,1\})$, $\text{XOR}(\{0,0\},\{0,1\})$, and $\text{XOR}(\{0,1\},\{0,0\})$. For XOR inputs with b bits, there are $2^{\text{b}} - 1$ other combinations of inputs that would make the same output. This means if $C = \text{XOR}(A, B)$ and input $A \in_R \{0, ..., 2^{\text{b}-1}\}$, an observer who learned C would gain no information about B.

At the same time, anyone who knows two of the elements in $\{A, B, C\}$, where $C = \text{XOR}(A, B)$, can calculate the third element, such as $A = \text{XOR}(B, C)$. XOR indicates if two elements are different or the same, so knowing C and B is enough to expose A. A careful examination of the truth table reveals this vital feature.[30]

[30] One interesting application of XOR (unrelated to Monero) is swapping two bit registers without a third register. We use the symbol $\oplus$ to indicate an XOR operation. $A \oplus A = 0$, so after three XOR operations between the registers: $\{A, B\} \rightarrow \{[A \oplus B], B\} \rightarrow \{[A \oplus B], B \oplus [A \oplus B]\} = \{[A \oplus B], A \oplus 0\} = \{[A \oplus B], A\} \rightarrow \{[A \oplus B] \oplus A, A\} = \{B, A\}$.

CHAPTER 3

Advanced Schnorr-like Signatures

A basic Schnorr signature has one signing key. As it happens, we can apply its core concepts to create a variety of progressively more complex signature schemes. One of those schemes, MLSAG, will be of central importance in Monero's transaction protocol.

3.1 Prove knowledge of a discrete logarithm across multiple bases

It is often useful to prove the same private key was used to construct public keys on different 'base' keys. For example, we could have a normal public key kG, and a Diffie-Hellman shared secret kR with some other person's public key (recall Section 2.3.3), where the base keys are G and R. As we will soon see, we can prove knowledge of the discrete log k in kG, prove knowledge of k in kR, *and* prove that k is the same in both cases (all without revealing k).

Non-interactive proof

Suppose we have a private key k, and d base keys $\mathcal{J} = \{J_1, ..., J_d\}$. The corresponding public keys are $\mathcal{K} = \{K_1, ..., K_d\}$. We make a Schnorr-like proof (recall Section 2.3.4) across all bases.[1] Assume the existence of a hash function $\mathcal{H}_n$ mapping to integers from 0 to $l - 1$.[2]

[1] While we say 'proof', it can be trivially made a signature by including a message $\mathtt{m}$ in the challenge hash. The terminology is loosely interchangeable in this context.

[2] In Monero, the hash function $\mathcal{H}_n(x) = \text{sc\_reduce32}(\mathit{Keccak}(x))$ where *Keccak* is the basis of SHA3 and sc_reduce32() puts the 256 bit result in the range 0 to $l - 1$ (although it should really be 1 to $l - 1$).

1. Generate random number $\alpha \in_R \mathbb{Z}_l$, and compute, for all $i \in (1, ..., d)$, αJ_i.
2. Calculate the challenge,
$$c = \mathcal{H}_n(\mathcal{J}, \mathcal{K}, [\alpha J_1], [\alpha J_2], ..., [\alpha J_d])$$
3. Define the response $r = \alpha - c * k$.
4. Publish the signature (c, r).

Verification

Assuming the verifier knows $\mathcal{J}$ and $\mathcal{K}$, he does the following.

1. Calculate the challenge:
$$c' = \mathcal{H}(\mathcal{J}, \mathcal{K}, [rJ_1 + c * K_1], [rJ_2 + c * K_2], ..., [rJ_d + c * K_d])$$
2. If $c = c'$ then the signer must know the discrete logarithm across all bases, and it's the same discrete logarithm in each case (as always, except with negligible probability).

Why it works

If instead of d base keys there was just one, this proof would clearly be the same as our original Schnorr proof (Section 2.3.4). We can imagine each base key in isolation to see that the multi-base proof is just a bunch of Schnorr proofs connected together. Moreover, by using only one challenge and response for all of those proofs, they must have the same discrete logarithm k. To get a single response that works for multiple keys the challenge would need to be known before defining an α for each key, but c is a function of α!

3.2 Multiple private keys in one proof

Much like a multi-base proof, we can combine many Schnorr proofs that use different private keys. Doing so proves we know all the private keys for a set of public keys, and reduces storage requirements by making just one challenge for all proofs.

Non-interactive proof

Suppose we have d private keys $k_1, ..., k_d$, and base keys $\mathcal{J} = \{J_1, ..., J_d\}$.[3] The corresponding public keys are $\mathcal{K} = \{K_1, ..., K_d\}$. We make a Schnorr-like proof for all keys simultaneously.

[3] There is no reason $\mathcal{J}$ can't contain duplicate base keys here, or for all base keys to be the same (e.g. G). Duplicates would be redundant for multi-base proofs, but now we are dealing with different private keys.

1. Generate random numbers $\alpha_i \in_R \mathbb{Z}_l$ for all $i \in (1, ..., d)$, and compute all $\alpha_i J_i$.
2. Calculate the challenge,
$$c = \mathcal{H}_n(\mathcal{J}, \mathcal{K}, [\alpha_1 J_1], [\alpha_2 J_2], ..., [\alpha_d J_d])$$
3. Define each response $r_i = \alpha_i - c * k_i$.
4. Publish the signature $(c, r_1, ..., r_d)$.

Verification

Assuming the verifier knows $\mathcal{J}$ and $\mathcal{K}$, he does the following.

1. Calculate the challenge:
$$c' = \mathcal{H}(\mathcal{J}, \mathcal{K}, [r_1 J_1 + c * K_1], [r_2 J_2 + c * K_2], ..., [r_d J_d + c * K_d])$$
2. If $c = c'$ then the signer must know the private keys for all public keys in $\mathcal{K}$ (except with negligible probability).

3.3 Spontaneous Anonymous Group (SAG) signatures

Group signatures are a way of proving a signer belongs to a group, without necessarily identifying him. Originally (Chaum in [49]), 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. These relied on a *group secret* which 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 inside companies.

Liu *et al.* presented a more interesting scheme in [85] building on the work of Rivest *et al.* in [119]. The authors detailed a group signature algorithm called LSAG characterized by three properties: *anonymity, linkability,* and *spontaneity.* Here we discuss SAG, the non-linkable version of LSAG, for conceptual clarity. We reserve the idea of linkability for later sections.

Schemes with anonymity and spontaneity are typically referred to as 'ring signatures'. In the context of Monero they will ultimately allow for unforgeable, signer-ambiguous transactions that leave currency flows largely untraceable.

Signature

Ring signatures are composed of a ring and a signature. Each *ring* is a set of public keys, one of which belongs to the signer and the rest of which are unrelated. The *signature* is generated with that ring of keys, and anyone verifying it would not be able to tell which ring member was the actual signer.

Our Schnorr-like 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 $K_\pi \in \mathcal{R}$, where π is a secret index.

1. Generate random number $\alpha \in_R \mathbb{Z}_l$ and fake responses $r_i \in_R \mathbb{Z}_l$ for $i \in \{1, 2, ..., n\}$ but excluding $i = \pi$.
2. Calculate
$$c_{\pi+1} = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [\alpha G])$$
3. For $i = \pi + 1, \pi + 2, ..., n, 1, 2, ..., \pi - 1$ calculate, replacing $n + 1 \to 1$,
$$c_{i+1} = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [r_i G + c_i K_i])$$
4. Define the real response r_π such that $\alpha = r_\pi + c_\pi k_\pi \pmod{l}$.

The ring signature contains the signature $\sigma(\mathfrak{m}) = (c_1, r_1, ..., r_n)$, 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 $\mathcal{R}$ (without necessarily knowing which one), and is done in the following manner:

1. For $i = 1, 2, ..., n$ iteratively compute, replacing $n + 1 \to 1$,
$$c'_{i+1} = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [r_i G + c_i K_i])$$
2. If $c_1 = c'_1$ then the signature is valid. Note that c'_1 is the last term calculated.

In this scheme we store $(1+n)$ integers and use n public keys.

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. Generate random numbers: α, r_1, r_3
2. Seed the signature loop:
$$c_3 = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [\alpha G])$$
3. Iterate:
$$c_1 = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [r_3 G + c_3 K_3])$$
$$c_2 = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [r_1 G + c_1 K_1])$$
4. Close the loop by responding: $r_2 = \alpha - c_2 k_2 \pmod{l}$

We can substitute α into c_3 to see where the word 'ring' comes from:

$$c_3 = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [(r_2 + c_2 k_2) G])$$
$$c_3 = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [r_2 G + c_2 K_2\ ])$$

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

1. We use r_1 and c_1 to compute
$$c_2' = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [r_1 G + c_1 K_1])$$
2. From when we made the signature, we see $c_2' = c_2$. With r_2 and c_2' we compute
$$c_3' = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [r_2 G + c_2' K_2])$$
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
$$c_1' = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [r_3 G + c_3' K_3])$$

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

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

The ring signature schemes discussed here on out display several properties that will be useful for producing confidential transactions.[4] Note that both 'signer ambiguity' and 'unforgeability' also apply to SAG signatures.

[4] Keep in mind that all robust signature schemes have security models which contain various properties. The properties mentioned here are perhaps most relevant to understanding the purpose of Monero's ring signatures, but are not a comprehensive overview of linkable ring signature properties.

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.[5] Monero uses this to obfuscate the origin of funds in each transaction.

Linkability If a private key is used to sign two different messages then the messages will become linked.[6] 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.[7] This is used to prevent theft of Monero funds by those not in possession of the appropriate private keys.

In the LSAG signature scheme [85], the owner of a private key could produce one anonymous unlinked signature per ring.[8] In this section we present an enhanced version of the LSAG algorithm where linkability is independent of the ring's decoy members.[9]

The modification was unraveled in [108] based on a publication by Adam Back [35] regarding the CryptoNote [136] ring signature algorithm (previously used in Monero, and now deprecated; see Section 8.1.2), which was in turn inspired by Fujisaki and Suzuki's work in [65].

Signature

As with SAG, let $\mathfrak{m}$ be the message to sign, $\mathcal{R} = \{K_1, K_2, ..., K_n\}$ a set of distinct public keys, and k_π the signer's private key corresponding to his public key $K_\pi \in \mathcal{R}$, where π is a secret index. Assume the existence of a hash function $\mathcal{H}_p$, which maps to curve points in EC.[10,11]

1. Calculate key image $\tilde{K} = k_\pi \mathcal{H}_p(K_\pi)$.[12]

[5] 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 ring size, or e.g. 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 track down real actors.

[6] The linkability property does not apply to non-signing public keys. That is, a ring member whose public key has been used in different ring signatures will not cause linkage.

[7] 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 legitimate signatures for chosen 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 [108] and [85].

[8] 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 member per ring." Signatures can be linked even if made for different messages.

[9] LSAG was discussed in the first edition of this report. [33]

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

[11] 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 $n_x G = \mathcal{H}_p(x)$. See a description of the algorithm in [107]. According to the CryptoNote whitepaper [136] its origin was this paper: [133]. src/ringct/rctOps.cpp `hash_to_p3()`

[12] In Monero it's important to use the hash to point function for key images instead of another base point so linearity doesn't lead to linking signatures created by the same address (even if for different one-time addresses). See [136] page 18.

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 $\tilde{K}$ and 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 $n + 1 \to 1$,
$$c'_{i+1} = \mathcal{H}_n(\mathfrak{m}, [r_i G + c_i K_i], [r_i \mathcal{H}_p(K_i) + c_i \tilde{K}])$$

3. If $c_1 = c'_1$ then the signature is valid.

In this scheme we store $(1+n)$ integers, have one EC key image, and use n public keys.

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) to $\tilde{K}$ 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.[13] This is because an EC point multiplied by its subgroup's order is zero.[14]

src/cryptonote_core/cryptonote_core.cpp `check_tx_inputs_key-images_domain()`

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:
$$\begin{aligned} c * (K + K^h) &= cK + cK^h \\ &= cK + 0 \end{aligned}$$

We can demonstrate correctness (i.e. 'how it works') in a similar way to the more simple SAG signature scheme.

Our description attempts to be faithful to the original explanation of bLSAG, which does not include $\mathcal{R}$ in the hash that calculates c_i. Including keys in the hash is known as 'key prefixing'. Recent research [77] suggests it may not be necessary, although adding the prefix is standard practice for similar signature schemes (LSAG uses key prefixing).

[13] 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 (and hence, possible subgroups) are either multiples of l (a prime) or divisors of h.

[14] 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) [63].

Linkability

Given two valid signatures that are different in some way (e.g. different fake responses, different messages, different overall ring members),

$$\sigma(\mathfrak{m}) = (c_1, r_1, ..., r_n) \text{ with } \tilde{K}, \text{ and}$$
$$\sigma'(\mathfrak{m}') = (c'_1, r'_1, ..., r'_{n'}) \text{ with } \tilde{K}',$$

if $\tilde{K} = \tilde{K}'$ then clearly both signatures come from the same private key.

While an observer could link σ and σ', he wouldn't necessarily know which K_i in $\mathcal{R}$ or $\mathcal{R}'$ was the culprit unless there was only one common key between them. If there was more than one common ring member, his only recourse would be solving the DLP or auditing the rings in some way (such as learning all k_i with $i \neq \pi$, or learning k_π).[15]

3.5 Multilayer Linkable Spontaneous Anonymous Group (MLSAG) signatures

In order to sign transactions, one has to sign with multiple private keys. In [108], Shen Noether *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 m private keys $\{k_{\pi,j}\}$ corresponding to the subset $\{K_{\pi,j}\}$ for some index $i = \pi$. Such an algorithm would address our needs if we generalize the notion of linkability.

Linkability If any private key $k_{\pi,j}$ is used in 2 different signatures, then those 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[16]

$$c_{\pi+1} = \mathcal{H}_n(\mathfrak{m}, [\alpha_1 G], [\alpha_1 \mathcal{H}_p(K_{\pi,1})], ..., [\alpha_m G], [\alpha_m \mathcal{H}_p(K_{\pi,m})])$$

[15] LSAG, which is quite similar to bLSAG, is unforgeable, meaning no attacker could make a valid ring signature without knowing a private key. If he invents a fake $\tilde{K}$ and seeds his signature computation with $c_{\pi+1}$, then, not knowing k_π, he can't calculate a number $r_\pi = \alpha - c_\pi k_\pi$ that would produce $[r_\pi G + c_\pi K_\pi] = \alpha G$. A verifier would reject his signature. Liu *et al.* prove forgeries that manage to pass verification are extremely improbable [85].

[16] Monero MLSAG uses key prefixing. Each challenge contains explicit public keys like this (adding the K terms absent from bLSAG; key images are included in the message signed):

$$c_{\pi+1} = \mathcal{H}_n(\mathfrak{m}, K_{\pi,1}, [\alpha_1 G], [\alpha_1 \mathcal{H}_p(K_{\pi,1})], ..., K_{\pi,m}, [\alpha_m G], [\alpha_m \mathcal{H}_p(K_{\pi,m})])$$

src/ringct/rctSigs.cpp `MLSAG_Gen()`

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

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

Verification

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 SAG 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,
  $$r_{\pi,j}G + c_\pi K_{\pi,j} = (\alpha_j - c_\pi k_{\pi,j})G + c_\pi K_{\pi,j} = \alpha_j G$$
  and
  $$r_{\pi,j}\mathcal{H}_p(K_{\pi,j}) + c_\pi \tilde{K}_j = (\alpha_j - c_\pi k_{\pi,j})\mathcal{H}_p(K_{\pi,j}) + c_\pi \tilde{K}_j = \alpha_j \mathcal{H}_p(K_{\pi,j})$$
  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.[17]

Space requirements

In this scheme we store $(1+m*n)$ integers, have m EC key images, and use $m*n$ public keys.

[17] As with bLSAG, linked MLSAG signatures do not indicate which public key was used to sign it. However, if the linking key image's sub-loops' rings have only one key in common, the culprit is obvious. If the culprit is identified, all other signing members of both signatures are revealed since they share the culprit's indices.

3.6 Concise Linkable Spontaneous Anonymous Group (CLSAG) signatures

CLSAG [67][18] is sort of half-way between bLSAG and MLSAG. Suppose you have a 'primary' key, and associated with it are several 'auxiliary' keys. It is important to prove knowledge of all private keys, but linkability only applies to the primary. This linkability retraction allows smaller, faster signatures than afforded by MLSAG.

As with MLSAG, we have a set of $n \cdot m$ keys (n is the ring size, m is the number of signing keys), and the primary keys are at index 1. In other words, there are n primary keys, and the π^{th} such key and its auxiliaries will sign.

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

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

Signature

1. Calculate key images $\tilde{K_j} = k_{\pi,j}\mathcal{H}_p(K_{\pi,1})$ for all $j \in \{1, 2, ..., m\}$. Note the base key is always the same, and so key images with $j > 1$ are 'auxiliary key images'. For notational simplicity we call them all $\tilde{K_j}$.
2. Generate random numbers $\alpha \in_R \mathbb{Z}_l$, and $r_i \in_R \mathbb{Z}_l$ for $i \in \{1, 2, ..., n\}$ (except $i = \pi$).
3. Calculate aggregate public keys W_i for $i \in \{1, 2, ..., n\}$, and aggregate key image $\tilde{W}$[19]
$$W_i = \sum_{j=1}^{m} \mathcal{H}_n(T_j, \mathcal{R}, \tilde{K_1}, ..., \tilde{K_m}) * K_{i,j}$$
$$\tilde{W} = \sum_{j=1}^{m} \mathcal{H}_n(T_j, \mathcal{R}, \tilde{K_1}, ..., \tilde{K_m}) * \tilde{K_j}$$
where $w_\pi = \sum_j \mathcal{H}_n(T_j, ...) * k_{\pi,j}$ is the aggregate private key.
4. Compute
$$c_{\pi+1} = \mathcal{H}_n(T_c, \mathcal{R}, \mathfrak{m}, [\alpha G], [\alpha\mathcal{H}_p(K_{\pi,1})])$$
5. For $i = \pi + 1, \pi + 2, ..., n, 1, 2, ..., \pi - 1$ calculate, replacing $n + 1 \to 1$,
$$c_{i+1} = \mathcal{H}_n(T_c, \mathcal{R}, \mathfrak{m}, [r_i G + c_i W_i], [r_i\mathcal{H}_p(K_{i,1}) + c_i\tilde{W}])$$
6. Define $r_\pi = \alpha - c_\pi w_\pi \pmod{l}$.

Therefore $\sigma(\mathfrak{m}) = (c_1, r_1, ..., r_n)$, with primary key image $\tilde{K_1}$, and auxiliary images $(\tilde{K_2}, ..., \tilde{K_m})$.

[18] The paper this section is based on is a pre-print being finalized for external review. CLSAG is promising as a replacement for MLSAG in future protocol versions, but has not been implemented, and might not be in the future.

[19] The CLSAG paper says to use different hash functions for domain separation, which we model by prefixing each hash with a tag [67], e.g. T_1 = "CLSAG_1", T_c = "CLSAG_c", etc. Domain separated hash functions have different outputs even with the same inputs. We also use key prefixing here (including $\mathcal{R}$, which has all the keys, in the hash). Domain separating is a new policy for Monero development, and will likely be done with all applications of hash functions added in the future (v13+). Historical uses of hash functions will probably be left alone.

Verification

The verification of a signature is done in the following manner:

1. For all $j \in \{1, ..., m\}$ check $l\tilde{K}_j \stackrel{?}{=} 0$.[20]
2. Calculate aggregate public keys W_i for $i \in \{1, 2, ..., n\}$, and aggregate key image $\tilde{W}$

$$W_i = \sum_{j=1}^{m} \mathcal{H}_n(T_j, \mathcal{R}, \tilde{K}_1, ..., \tilde{K}_m) * K_{i,j}$$

$$\tilde{W} = \sum_{j=1}^{m} \mathcal{H}_n(T_j, \mathcal{R}, \tilde{K}_1, ..., \tilde{K}_m) * \tilde{K}_j$$

3. For $i = 1, ..., n$ compute, replacing $n + 1 \to 1$,

$$c_{i+1} = \mathcal{H}_n(T_c, \mathcal{R}, \mathfrak{m}, [r_i G + c_i W_i], [r_i \mathcal{H}_p(K_{i,1}) + c_i \tilde{W}])$$

4. If $c_1 = c_1'$ then the signature is valid.

Why it works

The biggest danger in concise signatures like this is key cancellation, where the key images reported aren't legitimate, yet still sum to a legitimate aggregate value. This is where the aggregation coefficients $\mathcal{H}_n(T_j, \mathcal{R}, \tilde{K}_1, ..., \tilde{K}_m)$ come into play, locking down each key to its expected value. We leave tracing out the circular repercussions of faking a key image as an exercise to the reader (perhaps start by imagining those coefficients don't exist). Auxiliary key images are an artifact of proving the primary image is legitimate, since the aggregate private key w_π, which contains all the private keys, is applied to base point $\mathcal{H}_p(K_{\pi,1})$.

Linkability

If a private key $k_{\pi,1}$ is re-used to make any signature, the corresponding primary key image $\tilde{K}_1$ supplied in the signature will reveal it. Auxiliary key images are ignored, as they only exist to facilitate the 'Concise' part of CLSAG.

Space requirements

We store (1+n) integers, have m key images, and use $m * n$ public keys.

[20] In Monero we would only check $l * \tilde{K}_1 \stackrel{?}{=} 0$ for the primary key image. Auxiliary keys would be stored as $(1/8) * \tilde{K}_j$, and during verification multiplied by 8 (recall Section 2.3.1), which is more efficient. The method discrepancy is an implementation choice, since linkable key images are very important and so shouldn't be messed with aggressively, and the other method was employed in prior protocol versions.

CHAPTER 4

Monero Addresses

The ownership of digital currency stored in a blockchain is controlled by so-called 'addresses'. Addresses are sent money which only the address-holder can spend.[1]

More specifically, an address owns the 'outputs' from some transactions, which are like notes giving the address-holder spending rights to an 'amount' of money. Such a note might say "Address C now owns 5.3 XMR".

To spend an owned output, the address-holder references it as the input to a new transaction. This new transaction has outputs owned by other addresses (or by the sender's address, if the sender wants). A transaction's total input amount equals its total output amount, and once spent an owned output can't be respent. Carol, who has Address C, could reference that old output in a new transaction (e.g. "In this transaction I'd like to spend that old output.") and add a note saying "Address B now owns 5.3 XMR".

An address's balance is the sum of amounts contained in its unspent outputs.[2]

We discuss hiding the amount from observers in Chapter 5, the structure of transactions in Chapter 6 (which includes how to prove you are spending an owned and previously unspent output, without even revealing which output is being spent), and the money creation process and role of observers in Chapter 7.

[1] Except with negligible probability.

[2] Computer applications known as 'wallets' are used to find and organize the outputs owned by an address, to maintain custody of its private keys for authoring new transactions, and to submit those transactions to the network for verification and inclusion in the blockchain.

4.1 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).[3]

src/common/base58.cpp `encode_addr()`

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 their address owns an output, and their spend key will allow them to spend that output in a transaction (and retroactively figure out it has been spent).[4]

4.2 One-time addresses

To receive money, a Monero user may distribute their address to other users, who can then send it money via transaction outputs.

The address is never used directly.[5] Instead, a Diffie-Hellman-like exchange is applied to it, creating a unique *one-time address* for each transaction output to be paid to the user. In this way, even external observers who know all users' addresses cannot use them to identify which user owns any given transaction output.[6]

Let's start with a very simple mock transaction, containing exactly one output — a payment of '0' amount from Alice to Bob.

Bob has private/public keys (k_B^v, k_B^s) and (K_B^v, K_B^s), and Alice knows his public keys (his address). The mock transaction could proceed as follows [136]:

1. Alice generates a random number $r \in_R \mathbb{Z}_l$, and calculates the one-time address[7]

[3] To communicate an address to other users, it is extremely common to encode it in base-58, a binary-to-text encoding scheme first created for Bitcoin [28]. See [5] for more details.

[4] 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 (view and spend) all of the outputs they own (spent and unspent). 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 $k^s = \texttt{sc\_reduce32}(Keccak(a))$ and $k^v = \texttt{sc\_reduce32}(Keccak(Keccak(a)))$. [5]

src/wallet/api/wallet.cpp `create()` wallet2.cpp `generate()` `get_seed()`

[5] The method described here is not enforced by the protocol, just by wallet implementation standards. This means an alternate wallet could follow the style of Bitcoin where recipients' addresses are included directly in transaction data. Such a non-compliant wallet would produce transaction outputs unusable by other wallets, and each Bitcoin-esque address could only be used once due to key image uniqueness.

[6] Except with negligible probability.

[7] In Monero, every instance (including when it's used for other parts of the transaction) of rk^vG is multiplied by the cofactor 8, so in this case Alice computes $8 * rK_B^v$ and Bob computes $8 * k_B^v rG$. As far as we can tell this serves no purpose (but it *is* a rule that users must follow). Multiplying by the cofactor ensures the resulting point is in G's subgroup, but if R and K^v don't share a subgroup to begin with, then the discrete logs r and k^v can't be used to make a shared secret regardless.

src/crypto/crypto.cpp `generate_key_derivation()`

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

src/crypto/crypto.cpp `derive_public_key()`

2. Alice sets K^o as the addressee of the payment, adds the output amount '0' and 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 $k_B^v rG = rK_B^v$. He can then calculate $K_B'^s = K^o - \mathcal{H}_n(rK_B^v)G$. When he sees that $K_B'^s = K_B^s$, he knows the output is addressed to him.[8]

   The private key k_B^v is called the 'view key' because anyone who has it (and Bob's public spend key K_B^s) can calculate $K_B'^s$ for every transaction output in the blockchain (record of transactions), and 'view' which ones belong to Bob.

   src/crypto/crypto.cpp `derive_subaddress_public_key()`

4. The one-time keys for the output are:

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

To spend his '0' amount output [sic] in a new transaction, all Bob needs to do is prove ownership by signing a message with the one-time key K^o. The private key k_B^s is the 'spend key' since it is required for proving output ownership, while k_B^v is the 'view key' since it can be used to find outputs spendable by Bob.

As will become clear in Chapter 6, 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.[9]

Bob may give a third party his view key. 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 output's amount (to be explained in Section 5.3). See Chapter 8 for other ways Bob could provide information about his transaction history.

4.2.1 Multi-output transactions

Most transactions will contain more than one output. If nothing else, to transfer 'change' back to the sender.[10,11]

[8] $K_B'^s$ is computed with `derive_subaddress_public_key()` because normal address spend keys are stored at index 0 in the spend key lookup table, while subaddresses are at indices 1,2... This will make sense soon: see Section 4.3.

[9] 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_B^v, there is no reason she can't do it. Bob can only spend one of those outputs because each one-time 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.

[10] Actually, as of protocol v12 two outputs are *required* from each (non-miner) transaction, even if it means one output has 0 amount (`HF_VERSION_MIN_2_OUTPUTS`). This improves transaction indistinguishability by mixing 1-output cases with the much more common 2-output transactions. Previous to v12 the core wallet software was already creating 0-value outputs. The core implementation sends 0-amount outputs to a random address.

[11] After Bulletproofs were implemented in protocol v8, each transaction became limited to no more than 16 outputs (`BULLETPROOF_MAX_OUTPUTS`). Previously there was no limit, aside from a restraint on transaction size (in

src/wallet/wallet2.cpp `transfer_selected_rct()`

Monero senders usually generate only one random value r. The curve point rG is typically known as the *transaction public key* and is published alongside other transaction data in the blockchain.

src/cryptonote_core/cryptonote_tx_utils.cpp `construct_tx_with_tx_key()`

To ensure that all one-time 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 $t \in \{0, ..., p-1\}$. By appending this value to the shared secret before hashing it, one can ensure the resulting one-time addresses are unique:

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

src/crypto/crypto.cpp `derive_public_key()`

4.3 Subaddresses

Monero users can generate subaddresses from each address [105]. Funds sent to a subaddress can be viewed and spent using its main address's 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, most observers would have to solve the DLP to determine a given subaddress is derived from any particular address [105].[12]

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 $(K^{v,i}, K^{s,i})$:[13]

src/device/device_default.cpp `get_subaddress_secret_key()`

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

So,

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

bytes).

[12] Prior to subaddresses (added in the software update corresponding with protocol v7 [76]), Monero users could simply generate many normal addresses. To view each address's balance, you needed to do a separate scan of the blockchain record. This was very inefficient. With subaddresses, users maintain a look-up table of (hashed) spend keys, so one scan of the blockchain takes the same amount of time for 1 subaddress, or 10,000 subaddresses.

[13] It turns out the subaddress hash is domain separated, so it's really $\mathcal{H}_n(T_{sub}, k^v, i)$ where T_{sub} ="SubAddr". We omit T_{sub} throughout this document for brevity.

4.3.1 Sending to a subaddress

Let's say Alice is going to send Bob '0' amount again, this time via his subaddress $(K_B^{v,1}, K_B^{s,1})$.

1. Alice generates a random number $r \in_R \mathbb{Z}_l$, and calculates the one-time address
$$K^o = \mathcal{H}_n(rK_B^{v,1}, 0)G + K_B^{s,1}$$
2. Alice sets K^o as the addressee of the payment, adds the output amount '0' and 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 $K_B'^s = K^o - \mathcal{H}_n(rK_B^{v,1}, 0)G$. When he sees that $K_B'^s = K_B^{s,1}$, he knows the transaction is addressed to him.[14]

   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.

   src/crypto/crypto.cpp `derive_subaddress_public_key()`
4. The one-time keys for the output are:
$$K^o = \mathcal{H}_n(rK_B^{v,1}, 0)G + K_B^{s,1} = (\mathcal{H}_n(rK_B^{v,1}, 0) + k_B^{s,1})G$$
$$k^o = \mathcal{H}_n(rK_B^{v,1}, 0) + k_B^{s,1}$$

Now, Alice's 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 $t \in \{0, ..., p-1\}$. 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.[15,16]

src/cryptonote_core/cryptonote_tx_utils.cpp `construct_tx_with_tx_key()`

[14] An advanced attacker may be able to link subaddresses [56] (a.k.a. the Janus attack). With subaddresses (one of which can be a normal address) K_B^1 & K_B^2 the attacker thinks may be related, he makes a transaction output with $K^o = \mathcal{H}_n(rK_B^{v,2}, 0)G + K_B^{s,1}$ and includes transaction public key $rK_B^{s,2}$. Bob calculates $rK_B^{v,2}$ to find $K_B'^{s,1}$ but has no way of knowing it was his *other* (sub)address's key used! If he tells the attacker that he received funds to K_B^1, the attacker will know K_B^2 is a related subaddress (or normal address). Since subaddresses are outside the protocol's scope, mitigations are up to wallet implementers. No known wallets have done so, and any mitigation would only work for compliant wallets. Potential mitigations include: not informing attackers of received funds, including encrypted transaction private key r in transaction data, a Schnorr signature on the shared secret using $K^{s,1}$ as the base point instead of G, and including rG in transaction data and verifying the shared secret with $rK^{s,1} \stackrel{?}{=} (k^s + \mathcal{H}_n(k^v, 1)) * rG$ (requires the private spend key). Outputs received by a normal address should also be verified. See [101] for a discussion of the last mitigation listed.

[15] 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.

src/cryptonote_basic/cryptonote_basic_impl.cpp `get_account_address_as_str()`

[16] There is some intricacy to when additional transaction public keys are used (see the code path `transfer_selected_rct()` → `construct_tx_and_get_tx_key()` → `construct_tx_with_tx_key()` → `generate_output_ephemeral_keys()` and `classify_addresses()`) surrounding change outputs where the transaction author knows the recipient's view key (since it's himself; also the case for dummy change outputs, which are created when a 0-amount output is necessary, since authors generate those addresses). Whenever there are at least two non-change outputs, and at least one of their recipients is a subaddress, proceed in the normal way explained above (a current bug in the core implementation adds an extra transaction public key to transaction data even beyond the additional keys, which is not used for anything). If either just the change output is to a subaddress, or

4.4 Integrated addresses

To differentiate between the outputs they receive, a recipient can request senders include a *payment ID* in transaction data.[17] 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.

At one point senders could communicate payment IDs in clear text, but manually including the IDs in transactions is inconvenient, and cleartext is a privacy hazard for recipients, who might inadvertently expose their activities.[18] 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.[19]

src/cryptonote_basic/cryptonote_basic_impl.cpp `get_account_integrated_address_as_str()`

Senders addressing outputs to integrated addresses can encode payment IDs using the shared secret rK_t^v and an XOR operation (recall Section 2.5), which recipients can then decode with the appropriate transaction public key and another XOR operation [9]. Encoding payment IDs in this way also allows senders to prove they made particular transaction outputs (e.g. for audits, refunds, etc.).[20]

Encoding

The sender encodes the payment ID for inclusion in transaction data[21]

src/device/device_default.cpp `encrypt_payment_id()`

$$k_{\text{mask}} = \mathcal{H}_n(rK_t^v, \text{pid\_tag})$$
$$k_{\text{payment ID}} = k_{\text{mask}} \rightarrow \text{reduced to bit length of payment ID}$$
$$\text{encoded payment ID} = \text{XOR}(k_{\text{payment ID}}, \text{payment ID})$$

there is just one non-change output and it's to a subaddress, then only one transaction public key is created. In the former case, the transaction public key is rG and the change's one-time key is (subaddress index 1, using c subscript to denote the change's keys) $K^o = \mathcal{H}_n(k_c^v rG, t)G + K_c^{s,1}$ using the normal view key and subaddress's spend key. In the latter case the transaction public key $rK^{v,1}$ is based off the subaddress's view key, and the change's one time key is $K^o = \mathcal{H}_n(k_c^v * rK^{v,1}, t)G + K_c^s$. These details help mingle a portion of subaddress transactions amongst the more common normal address transactions, and 2-output transactions which compose around 95% of transaction volume as of this writing.

[17] Currently, Monero implementations only support one payment ID per transaction regardless of how many outputs it has.

[18] These long-form (256 bit) cleartext payment IDs were deprecated in v0.15 of the core software implementation (coincident with protocol v12 in November 2019). While other wallets may still support them and allow their inclusion in transaction data, the core wallet will ignore them.

[19] Within the authors' knowledge, integrated addresses have only ever been implemented for normal addresses.

[20] Since an observer can recognize the difference between transactions with and without payment IDs, using them is thought to make the Monero transaction history less uniform. Because of this, since protocol v10 the core implementation adds a dummy encrypted payment ID to all 2-output transactions. Decoding one will reveal all 0s (this is not required by the protocol, just best practice).

src/cryptonote_core/cryptonote_tx_utils.cpp `construct_tx_with_tx_key()`

[21] In Monero payment IDs for integrated addresses are conventionally 64 bits long.

We include pid_tag to ensure k_{mask} is different from the component $\mathcal{H}_n(rK_t^v, t)$ in one-time output addresses.[22]

Decoding

Whichever recipient the payment ID was created for can find it using his view key and the transaction public key rG:[23]

src/device/device.hpp `decrypt_payment_id()`

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

Similarly, senders can decode payment IDs they had previously encoded by recalculating the shared secret.

4.5 Multisignature addresses

Sometimes it is useful to share ownership of funds between different people/addresses. We dedicate Chapter 9 to elaborating this topic.

[22] In Monero, pid_tag = `ENCRYPTED_PAYMENT_ID_TAIL` = 141. In, for example, multi-input transactions we compute $\mathcal{H}_n(rK_t^v, t) \pmod{l}$ to ensure we are using a scalar less than the EC subgroup order, but since l is 253 bits and payment IDs are only 64 bits, taking the modulus for encoding payment IDs would be meaningless, so we don't.

[23] Transaction data does not indicate which output a payment ID 'belongs' to. Recipients have to identify their own payment IDs.

CHAPTER 5

Monero Amount Hiding

In most cryptocurrencies like Bitcoin, transaction output notes, which give spending rights to 'amounts' of money, communicate those amounts in clear text. This allows observers to easily verify the amount spent equals the amount sent.

In Monero we use *commitments* to hide output amounts from everyone except senders and receivers, while still giving observers confidence that a transaction sends no more or less than what is spent. As we will see, amount commitments must also have corresponding 'range proofs' which prove the hidden amount is within a legitimate range.

5.1 Commitments

Generally speaking, a cryptographic *commitment scheme* is a way of committing to a value without revealing the value itself. After committing to something, you are stuck with it.

For example, in a coin-flipping game Alice could privately commit to one outcome (i.e. 'call it') by hashing her committed value with secret data and publishing the hash. After Bob flips the coin, Alice declares which value she committed to and proves it by revealing the 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 hashes $h = \mathcal{H}(blah, heads)$ and gives h to Bob. Bob flips a coin, then Alice tells Bob the secret string *blah* and that she committed to *heads*. 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*.[1]

5.2 Pedersen commitments

A *Pedersen commitment* [113] 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 $C(a+b) = C(a) + C(b)$.[2] This property will be useful when committing transaction amounts, as one can 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 could easily create cheat tables of commitments to help us recognize common values of 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.[3]

We can then define the commitment to a value a as $C(x, a) = xG + aH$, where x is the blinding factor (a.k.a. 'mask') that prevents observers from guessing a.

Commitment $C(x, a)$ is information-theoretically private because there are many possible combinations of x and a that would output the same C.[4] If x is truly random, an attacker would have literally no way to figure out a [88, 122].

5.3 Amount commitments

In Monero, output amounts are stored in transactions as Pedersen commitments. We define a commitment to an output's amount b as:

[1] If the committed value is very difficult to guess and check, e.g. if it's an apparently random elliptic curve point, then salting the commitment isn't necessary.

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

[3] In the case of Monero, $H = 8 * to\_point(\mathcal{H}_n(G))$. This differs from the $\mathcal{H}_p$ hash function in that it directly interprets the output of $\mathcal{H}_n(G)$ as a compressed point coordinate instead of deriving a curve point mathematically (see [107]). The historical reasons for this discrepancy are unknown to us, and indeed this is the only place where $\mathcal{H}_p$ is not used (Bulletproofs also use $\mathcal{H}_p$). Note how there is a $*8$ operation, to ensure the resultant point is in our l subgroup ($\mathcal{H}_p$ also does that). src/ringct/rctTypes.h tests/unit_tests/ringct.cpp `TEST(ringct, HPow2)`

[4] 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. This property is also known as 'perfect hiding' [138]. Furthermore, even the committer can't find another combination without solving the DLP for γ, a property called 'computational binding' [138].

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

src/ringct/rctOps.cpp `addKeys2()`

Recipients should be able to know how much money is in each output they own, as well as reconstruct the amount commitments, so they can be used as the inputs to new transactions. This means the blinding factor y and amount b must be communicated to the receiver.

The solution adopted is a Diffie-Hellman shared secret rK_B^v using the 'transaction public key' (recall Section 4.2.1). For any given transaction in the blockchain, each of its outputs $t \in \{0, ..., p-1\}$ has a mask y_t that senders and receivers can privately compute, and an *amount* stored in the transaction's data. While y_t is an elliptic curve scalar and occupies 32 bytes, b will be restricted to 8 bytes by the range proof so only an 8 byte value needs to be stored.[5,6]

$$y_t = \mathcal{H}_n(\text{“}commitment\_mask\text{”}, \mathcal{H}_n(rK_B^v, t))$$
$$amount_t = b_t \oplus_8 \mathcal{H}_n(\text{“}amount\text{”}, \mathcal{H}_n(rK_B^v, t))$$

src/ringct/rctOps.cpp `ecdhEncode()`

Here, $\oplus_8$ means to perform an XOR operation (Section 2.5) between the first 8 bytes of each operand (b_t which is already 8 bytes, and $\mathcal{H}_n(...)$ which is 32 bytes). Recipients can perform the same XOR operation on $amount_t$ to reveal b_t.

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

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

5.4 RingCT introduction

A transaction will contain references to other transactions' outputs (telling observers which old outputs are to be spent), and its own outputs. The content of an output includes a one-time address (assigning ownership of the output) and an output commitment hiding the amount (also the encoded output amount from Section 5.3).

While a transaction's verifiers don't know how much money is contained in each input and output, they still need to be sure the sum of input amounts equals the sum of output amounts. Monero uses a technique called RingCT [108], first implemented in January 2017 (v4 of the protocol), to accomplish this.

[5] As with the one-time address K^o from Section 4.2, the output index t is appended to the shared secret before hashing. This ensures outputs directed to the same address do not have similar masks and *amounts*, except with negligible probability. Also like before, the term rK_B^v is multiplied by 8, so it's really $8rK_B^v$.

[6] This solution (implemented in v10 of the protocol) replaced a previous method that used more data, thereby causing the transaction type to change from v3 (`RCTTypeBulletproof`) to v4 (`RCTTypeBulletproof2`). The first edition of this report discussed the previous method [33].

src/cryptonote_core/cryptonote_tx_utils.cpp `construct_tx_with_tx_key()` calls `generate_output_ephemeral_keys()`

If we have a transaction with m inputs containing amounts $a_1, ..., a_m$, and p outputs with amounts $b_0, ..., b_{p-1}$, then an observer would justifiably expect that:[7]

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

Since commitments are additive and we don't know γ, we could easily prove our inputs equal outputs to observers by making the sum of commitments to input and output amounts equal zero (i.e. by setting the sum of output blinding factors equal to the sum of old input blinding factors):[8]

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

To avoid sender identifiability we use a slightly different approach. The amounts being spent correspond to the outputs of previous transactions, which 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*, since she can make a signature with the private key $(x_j - x_j') = z_j$ and prove there is no H component to the sum (assuming γ is unknown). In other words prove that $C_j^a - C_j'^a = z_j G + 0H$, which we will actually do in Chapter 6 when we discuss the structure of RingCT transactions.

Let us call $C_j'^a$ a *pseudo output commitment*. Pseudo output commitments are included in transaction data, and there is one for each input.

Before committing a transaction to the blockchain, the network will want to verify that its amounts balance. Blinding factors for pseudo and output commitments are selected such that

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

This allows us to prove input amounts equal output amounts:

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

src/ringct/rctSigs.cpp verRctSemanticsSimple()

Fortunately, choosing such blinding factors is easy. In the current version of Monero, all blinding

[7] If the intended total output amount doesn't precisely equal any combination of owned outputs, then transaction authors can add a 'change' output sending extra money back to themselves. By analogy to cash, with a 20$ bill and 15$ expense you will receive 5$ back from the cashier.

[8] Recall from Section 2.3.1 we can subtract a point by inverting its coordinates then adding it. If $P = (x, y)$, $-P = (-x, y)$. Recall also that negations of field elements are calculated (mod q), so ($-x$ (mod q)).

factors are random except for the m^{th} pseudo out commitment, where x'_m is simply genRctSimple()

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

5.5 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

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

Since $-10 = l - 10$, we have effectively created l more Moneroj (over 7.2x10^{74}!) than we put in.

The solution addressing this issue in Monero is to prove each output amount is in a certain range (from 0 to $2^{64} - 1$) using the Bulletproofs proving scheme first described by Benedikt Bünz *et al.* in [43] (and also explained in [138, 50]).[9] Given the involved and intricate nature of Bulletproofs, it is not elucidated in this document. Moreover we feel the cited materials adequately present its concepts.[10]

The Bulletproof proving algorithm takes as input output amounts b_t and commitment masks y_t, and outputs all C_t^b and an n-tuple aggregate proof $\Pi_{BP} = (A, S, T_1, T_2, \tau_x, \mu, \mathbb{L}, \mathbb{R}, a, b, t)$[11,12]. That single proof is used to prove all output amounts are in range at the same time, as aggregating them greatly reduces space requirements (although it does increase the time to verify).[13] The verification algorithm takes as input all C_t^b, and Π_{BP}, and outputs `true` if all committed amounts are in the range 0 to $2^{64} - 1$. src/ringct/rctSigs.cpp `proveRangeBulletproof()`

The n-tuple Π_{BP} occupies $(2 \cdot \lceil \log_2(64 \cdot p) \rceil + 9) \cdot 32$ bytes of storage.

[9] It's conceivable that with several outputs in a legitimate range, the sum of their amounts could roll over and cause a similar problem. However, when the maximum output is much smaller than l it takes a huge number of outputs for that to happen. For example, if the range is 0-5, and l = 99, then to counterfeit money using an input of 2, we would need $5 + 5 + \ldots + 5 + 1 = 101 \equiv 2 \pmod{99}$, for 21 total outputs. In Monero l is about 2^189 times bigger than the available range, which means a ridiculous 2^189 outputs to counterfeit money.

[10] Prior to protocol v8 range proofs were accomplished with Borromean ring signatures, which were explained in the first edition of Zero to Monero [33].

[11] Vectors $\mathbb{L}$ and $\mathbb{R}$ contain $\lceil \log_2(64 \cdot p) \rceil$ elements each. $\lceil\ \rceil$ means the log function is rounded up. Due to their construction, some Bulletproofs use 'dummy outputs' as padding to ensure p plus the number of dummy outputs is a power of 2. Those dummy outputs can be generated during verification, and are not stored with the proof data.

[12] The variables in a Bulletproof are unrelated to other variables in this document. Symbol overlap is merely coincidental. Note that group elements $A, S, T_1, T_2, \mathbb{L}$, and $\mathbb{R}$ are multiplied by 1/8 before being stored, then multiplied by 8 during verification. This ensures they are all members of the l sub-group (recall Section 2.3.1).

[13] It turns out multiple separate Bulletproofs can be 'batched' together, which means they are verified simultaneously. Doing so improves how long it takes to verify them, and currently in Monero Bulletproofs are batched on a per-block basis, although there is no theoretical limit to how many can be batched together. Each transaction is only allowed to have one Bulletproof. rct/ringct/bulletproofs.cc `bulletproof_VERIFY()`

CHAPTER 6

Monero Ring Confidential Transactions (RingCT)

Throughout Chapters 4 and 5 we built up several aspects of Monero transactions. At this point a simple one-input, one-output transaction from some unknown author to some unknown recipient sounds like:

"My transaction uses transaction public key rG. I will spend old output X (note that it has a hidden amount A_X, committed to in C_X). I will give it a pseudo output commitment C'_X. I will make one output Y, which may be spent by the one-time address K^o_Y. It has a hidden amount A_Y committed to in C_Y, encrypted for the recipient, and proven in range with a Bulletproofs-style range proof. Please note that $C'_X - C_Y = 0$."

Some questions remain. Did the author actually own X? Does the pseudo output commitment C'_X actually correspond to C_X, such that $A_X = A'_X = A_Y$? Has someone tampered with the transaction, and perhaps directed the output at a recipient unintended by the original author?

As mentioned in Section 4.2, we can prove ownership of an output by signing a message with its one-time address (whoever has the address's key owns the output). We can also prove it has the same amount as a pseudo output commitment by proving knowledge of the commitment to zero's private key ($C_X - C'_X = z_X G$). Moreover, if that message is *all the transaction data* (except the signature itself), then verifiers can be assured everything is as the author intended (the signature only works with the original message). MLSAG signatures allow us to do all of this while also obscuring the actual spent output amongst other outputs from the blockchain, so observers can't be sure which one is being spent.

6.1 Transaction types

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

src/cryptonote_core/cryptonote_tx_utils.cpp `construct_tx_with_tx_key()`

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 schemes, even if they are still partially supported.[1] The transaction type we have discussed so far, and which will be further elaborated in this chapter, is `RCTTypeBulletproof2`.[2]

We present a conceptual summary of transactions in Section 6.3.

6.2 Ring Confidential Transactions of type `RCTTypeBulletproof2`

Currently (protocol v12) all new transactions must use this transaction type, in which each input is signed separately. An actual example of an `RCTTypeBulletproof2` transaction, with all its components, can be inspected in Appendix A.

6.2.1 Amount commitments and transaction fees

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 4.2). The sender also knows the blinding factors x_j used in commitments $C^a_{\pi,j}$ (Section 5.3).

Typically transaction outputs are *lower* in total than transaction inputs, in order to provide a fee that will incentivize miners to include the transaction in the blockchain.[3] Transaction fee amounts f are stored in clear text in the transaction data transmitted to the network. Miners can create an additional output for themselves with the fee (see Section 7.3.6).

[1] RingCT was first implemented in January 2017 (v4 of the protocol). It was made mandatory for all new transactions in September 2017 (v6 of the protocol) [16]. RingCT is version 2 of Monero's transaction protocol.

[2] Within the RingCT era there are three deprecated transaction types: `RCTTypeFull`, `RCTTypeSimple`, and `RCTTypeBulletproof`. The former two coexisted in the first iteration of RingCT and are explored in the first edition of this report [33], then with the advent of Bulletproofs (protocol v8) `RCTTypeFull` was deprecated, and `RCTTypeSimple` was upgraded to `RCTTypeBulletproof`. `RCTTypeBulletproof2` arrived due to a minor improvement in encrypting output commitments' masks and amounts (v10).

[3] In Monero there is a minimum base fee that scales with transaction weight. It is semi-mandatory because while you can create new blocks containing tiny-fee transactions, most Monero nodes won't relay such transactions to other nodes. The result is few if any miners will try to include them in blocks. Transaction authors can provide miner fees above the minimum if they want. We go into more detail on this in Section 7.3.4.

A transaction consists of inputs $a_1, ..., a_m$ and outputs $b_0, ..., b_{p-1}$ such that $\sum_{j=1}^{m} a_j - \sum_{t=0}^{p-1} b_t - f = 0$.[4]

The sender calculates pseudo output commitments for the input amounts, $C'^a_{\pi,1}, ..., C'^a_{\pi,m}$, and creates commitments for intended output amounts $b_0, ..., b_{p-1}$. Let these new commitments be $C^b_0, ..., C^b_{p-1}$.

He knows the private keys $z_1, ..., z_m$ to the commitments to zero $(C^a_{\pi,1} - C'^a_{\pi,1}), ..., (C^a_{\pi,m} - C'^a_{\pi,m})$.

For verifiers to confirm transaction amounts sum to zero, the fee amount must be converted into a commitment. The solution is to calculate the commitment of the fee f without the masking effect of any blinding factor. That is, $C(f) = fH$.

Now we can prove input amounts equal output amounts:

src/ringct/rctSigs.cpp verRctSemanticsSimple()

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

6.2.2 Signature

The sender selects m sets of size v, of additional unrelated one-time addresses and their commitments from the blockchain, corresponding to apparently unspent outputs.[5,6] To sign input j, she stirs a set of size v into a *ring* with her own j^{th} unspent one-time address (placed at unique index π), along with false commitments to zero, as follows:[7]

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

src/cryptonote_core/cryptonote_tx_utils.cpp `construct_tx_with_tx_key()`

src/wallet/wallet2.cpp `get_outs()` `gamma_picker::pick()`

[4] Outputs are randomly shuffled by the core implementation before getting assigned an index, so observers can't build heuristics around their order. Inputs are sorted by key image within transaction data.

[5] In Monero it is standard for the sets of 'additional unrelated addresses' to be selected from a gamma distribution across the range of historical outputs (RingCT outputs only, a triangle distribution is used for pre-RingCT outputs). This method uses a binning procedure to smooth out differences in block densities. First calculate the average time between transaction outputs over up to a year ago of RingCT outputs (avg time = [#outputs/#blocks]*blocktime). Select an output via the gamma distribution, then look inside its block and grab a random output to be a member of the set. [93]

[6] As of protocol v12, all transaction inputs must be at least 10 blocks old (`CRYPTONOTE_DEFAULT_TX_SPENDABLE_AGE`). Prior to v12 the core implementation used 10 blocks by default, but it was not required so an alternate wallet could make different choices, and some apparently did [82].

[7] In Monero each of a transaction's rings must have the same size, and the protocol controls how many ring members there can be per output-to-be-spent. It has changed with different protocol versions: v2 March 2016 ≥ 3, v6 Sept. 2017 ≥ 5, v7 April 2018 ≥ 7, v8 Oct. 2018 11-only. Since v6 each individual ring can not contain duplicate members, though there may be duplicates between rings to allow multiple inputs when there are insufficient total outputs in the transaction history (i.e. to assemble rings without cross-over) [135].

src/cryptonote_core/cryptonote_core.cpp `check_tx_inputs_ring_members_diff()`

Alice uses an MLSAG signature (Section 3.5) to sign this ring, where she knows the private keys $k^o_{\pi,j}$ for $K^o_{\pi,j}$, and z_j for the commitment to zero ($C^a_{\pi,j}$ - $C'^a_{\pi,j}$). Since no key image is needed for the commitments to zero, there is consequently no corresponding key image component in the signature's construction.[8]

src/ringct/rctSigs.cpp proveRctMGSimple()

Each input in a transaction is signed individually using rings like $\mathcal{R}_j$ as defined above, thereby obscuring the real outputs being spent, $(K^o_{\pi,1}, ..., K^o_{\pi,m})$, amongst other unspent outputs.[9] Since part of each ring includes a commitment to zero, the pseudo output commitment used must contain an amount equal to the real input being spent. This ties input amounts to the proof that amounts balance, without compromising which ring member is the real input.

The message $\mathfrak{m}$ signed by each input is essentially a hash of all transaction data *except* for the MLSAG signatures.[10] This ensures transactions are tamper-proof from the perspective of both transaction authors and verifiers. Only one message is produced, and each input MLSAG signs it.

One-time private key k^o is the essence of Monero's transaction model. Signing $\mathfrak{m}$ with k^o proves you are the owner of the amount committed to in C^a. Verifiers can be confident that transaction authors are spending their own funds, without even knowing which funds are being spent, how much is being spent, or what other funds they might own!

6.2.3 Avoiding double-spending

An MLSAG signature (Section 3.5) contains images $\tilde{K}_j$ of private keys $k_{\pi,j}$. An important property for 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.

src/cryptonote_core/blockchain.cpp `have_tx_keyimges_as_spent()`

The network only needs to verify that key images included with MLSAG signatures (corresponding to inputs and calculated as $\tilde{K}^o_j = k^o_{\pi,j}\mathcal{H}_p(K^o_{\pi,j})$) have not appeared before in other transactions.[11] If they have, then we can be sure we are witnessing an attempt to re-spend an output $(C^a_{\pi,j}, K^o_{\pi,j})$.

[8] Building and verifying the signature excludes the term $r_{i,2}\mathcal{H}_p(C_{i,j} - C'^a_{\pi,j}) + c_i\tilde{K}_{z_j}$.

[9] 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 would be in the aggregated case. This means even if one input's origin becomes identifiable, the other inputs' origins would not. The old transaction type `RCTTypeFull` used aggregated ring signatures, combining all rings into one, and as we understand it was deprecated for that reason.

[10] The actual message is $\mathfrak{m} = \mathcal{H}(\mathcal{H}(tx\_prefix), \mathcal{H}(ss), \mathcal{H}(\text{range proofs}))$ where:
tx_prefix ={transaction era version (i.e. RingCT = 2), inputs {ring member key offsets, key images}, outputs {one-time addresses}, extra {transaction public key, payment ID or encoded payment ID, misc.}}
ss ={transaction type (`RCTTypeBulletproof2` = '4'), transaction fee, pseudo output commitments for inputs, ecdhInfo (encrypted amounts), output commitments}.
See Appendix A regarding this terminology.

src/ringct/rctSigs.cpp `get_pre_mlsag_hash()`

[11] Verifiers must also check the key image is a member of the generator's subgroup (recall Section 3.4).

6.2.4 Space requirements

MLSAG signature (inputs)

From Section 3.5 we recall that an MLSAG signature in this context would be expressed as

$$\sigma_j(\mathfrak{m}) = (c_1, r_{1,1}, r_{1,2}, ..., r_{v+1,1}, r_{v+1,2}) \text{ with } \tilde{K}_j^o$$

As a legacy of 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 (Section 2.4.2), since each ring $\mathcal{R}_j$ contains $(v+1)\cdot 2$ keys, an input signature σ_j will require $(2(v+1)+1)\cdot 32$ bytes. On top of this, the key image $\tilde{K}_{\pi,j}^o$ and the pseudo output commitment $C_{\pi,j}'^a$ leave a total of $(2(v+1)+3)\cdot 32$ bytes per input.

To this value we would need 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.[12,13,14]

src/ringct/rctSigs.cpp verRctMGSimple() verRctSemanticsSimple()

Verifying all of an `RCTTypeBulletproof2` transaction's MLSAGs includes the computation of $(C_{i,j} - C_{\pi,j}'^a)$ and $(\sum_j C_j'^a \stackrel{?}{=} \sum_t C_t^b + fH)$, and verifying key images are in G's subgroup with $l\tilde{K} \stackrel{?}{=} 0$.

Range proofs (outputs)

An aggregate Bulletproof range proof will require $(2\cdot\lceil\log_2(64\cdot p)\rceil + 9)\cdot 32$ total bytes.

src/ringct/bulletproofs.cpp `bulletproof_VERIFY()`

[12] See [66] or [24] for an explanation of Monero's varint data type. It is an integer type that uses up to 9 bytes, and stores up to 63 bits of information.

src/common/varint.h

[13] Imagine the blockchain contains a long list of transaction outputs. We report the indices of outputs we want to use in rings. Now, bigger indexes require more storage space. We just need the 'absolute' position of *one* index from each ring, and the 'relative' positions of the other ring members' indices. For example, with real indices {7,11,15,20} we just need to report {7,4,4,5}. Verifiers can compute the last index with (7+4+4+5 = 20). Ring members are organized in ascending order of blockchain index within rings.

src/cryptonote_basic/cryptonote_format_utils.cpp

[14] A transaction with 10 inputs using rings with 11 total members will need $((11\cdot 2+3)\cdot 32)\cdot 10 = 8000$ bytes for its inputs, with around 110 to 330 bytes for the offsets (there are 110 ring members).

`absolute_output_offsets_to_relative()`

6.3 Concept summary: Monero transactions

To summarize this chapter, and the previous two chapters, we present the main content of a transaction, organized for conceptual clarity. A real example can be found in Appendix A.

- Type: '0' is `RCTTypeNull` (for miners), '4' is `RCTTypeBulletproof2`
- 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_j terms c_1, and $r_{i,1}$ & $r_{i,2}$ for $i \in \{1, ..., v+1\}$
  - **Key image**: the key image $\tilde{K}_j^{o,a}$ for input j
  - **Pseudo output commitment**: $C_j'^a$ for input j
- Outputs: for each output $t \in \{0, ..., p-1\}$ to address or subaddress (K_t^v, K_t^s)
  - **One-time address**: $K_t^{o,b}$ for output t
  - **Output commitment**: C_t^b for output t
  - **Encoded amount**: so output owners can compute b_t for output t
    - *Amount*: $b_t \oplus_8 \mathcal{H}_n(\text{“}amount\text{”}, \mathcal{H}_n(rK_B^v, t))$
  - **Range proof**: an aggregate Bulletproof for all output amounts b_t
    - *Proof*: $\Pi_{BP} = (A, S, T_1, T_2, \tau_x, \mu, \mathbb{L}, \mathbb{R}, a, b, t)$
- Transaction fee: communicated in clear text multiplied by 10^{12} (i.e. atomic units, see Section 7.3.1), so a fee of 1.0 would be recorded as 1000000000000
- Extra: includes the transaction public key rG, or, if at least one output is directed to a subaddress, $r_t K_t^{s,i}$ for each subaddress'd output t and $r_t G$ for each normal address'd output t, and maybe an encoded payment ID (should be at most one per transaction)[15]

Our final one-input/one-output example transaction sounds like this: "My transaction uses transaction public key rG. I will spend one of the outputs in set $\mathbb{X}$ (note that it has a hidden amount A_X, committed to in C_X). The output being spent is owned by me (I made a MSLAG signature on the one-time addresses in $\mathbb{X}$), and hasn't been spent before (its key image $\tilde{K}$ has not yet appeared in the blockchain). I will give it a pseudo output commitment C'_X. I will make one output Y, which may be spent by the one-time address K_Y^o. It has a hidden amount A_Y committed to in C_Y, encrypted for the recipient, and proven in range with a Bulletproofs-style range proof. My transaction includes a transaction fee f. Please note that $C'_X - (C_Y + C_f) = 0$, and that I have signed the commitment to zero $C'_X - C_X = zG$ which means the input amount equals the output amount $(A_X = A'_X = A_Y + f)$. My MLSAG signed all transaction data, so observers can be sure it hasn't been tampered with."

[15] No information stored in the 'extra' field is verified, though it *is* signed by input MLSAGs, so no tampering is possible (except with negligible probability). The field has no limit on how much data it can store, so long as the maximum transaction weight is respected. See [81] for more details.

6.3.1 Storage requirements

For `RCTTypeBulletproof2` we need $(2(v+1)+2) \cdot m \cdot 32$ bytes of storage, and the aggregate Bulletproof range proof needs $(2 \cdot \lceil \log_2(64 \cdot p) \rceil + 9) \cdot 32$ bytes of storage.[16]

Miscellaneous requirements:

- Input key images: $m * 32$ bytes
- One-time output addresses: $p * 32$ bytes
- Output commitments: $p * 32$ bytes
- Encoded output amounts: $p * 8$ 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. There should be no more than one per tx.
- Transaction fee: stored as a variable length integer, so ≤ 9 bytes
- Input offsets: stored as variable length integers, so ≤ 9 bytes per offset, for $m * (v+1)$ ring members
- Unlock time: stored as a variable length integer, so ≤ 9 bytes[17]
- 'Extra' tags: each piece of data in the 'extra' field (e.g. a transaction public key) begins with a 1 byte 'tag', and some pieces also have a 1+ byte 'length'; see Appendix A for more details

[16] The amount of transaction content is limited by a maximum so-called 'transaction weight'. Before Bulletproofs were added in protocol v8 (and indeed currently when transactions have only two outputs) the transaction weight and size in bytes were equivalent. The maximum weight is (0.5*300kB - `CRYPTONOTE_COINBASE_BLOB_RESERVED_SIZE`), where the blob reserve (600 bytes) is intended to leave room for the miner transaction within blocks. Before v8 the 0.5x multiplier was not included, and the 300kB term was smaller in earlier protocol versions (20kB v1, 60kB v2, 300kB v5). We elaborate on these topics in Section 7.3.2.

src/cryptonote_core/tx_pool.cpp `get_transaction_weight_limit()`

[17] Any transaction's author can lock its outputs, rendering them unspendable until a specified block height where it may be spent (or until a UNIX timestamp, at which time its outputs may be included in a block's transaction's ring members). He only has the option to lock all outputs to the same block height. It is not clear if this offers any meaningful utility to transaction authors (perhaps smart contracts). Miner transactions have a mandatory 60-block lock time. As of protocol v12 normal outputs can't be spent until after the default spendable age (10 blocks) which is functionally equivalent to a mandatory minimum 10-block lock time. If a transaction is published in the 10^{th} block with an unlock time of 25, it may be spent in the 25^{th} block or later. Unlock time is probably the least used of all transaction features for normal transactions.

src/cryptonote_core/blockchain.cpp `is_tx_spendtime_unlocked()`

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 [96], 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 [96]. Throughout history there have been many kinds of money, from seashells 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. [97]

To alleviate these drawbacks decentralized digital currencies have been engineered.[1]

7.1 Digital currency

Designing a digital currency is non-trivial. There are three types: personal, centralized, or distributed. Keep in mind that a digital currency is just a collection of messages, and the 'amounts' recorded in those messages are interpreted as monetary quantities.

[1] This chapter includes more implementation details than previous chapters, as a blockchain's nature depends heavily on its specific structure.

In the **email model** anyone can make coins (e.g. a message saying 'I own 5 coins'), and anyone can send their coins over and over to whoever has an email address. It does not have a limited supply, nor does it prevent spending the same coins over and over (double spending).

In the **video game model**, where the entire currency is stored/recorded 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 Distributed/shared version of events

In digital 'shared' money, many computers each have a record of every currency transaction. When a new transaction is made on one computer it is broadcast to the other computers, and accepted if it follows predefined rules.

Users only benefit from coins when other users accept them in exchange, and users only accept coins they feel are legitimate. To maximize the utility of their coins, users are naturally inclined to settle on one commonly accepted rule-set, without the presence of a central authority.[2]

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

Rule 2: Transactions spend money that already exists.

Rule 3: A person can only spend a piece of money once.

Rule 4: Only the person who owns a piece of money can spend it.

Rule 5: Transactions output money equal to the money spent.

Rule 6: Transactions are formatted correctly.

Rules 2-6 are covered by the transaction scheme discussed in Chapter 6, which adds the fungibility and privacy-related benefits of ambiguous signing, anonymous receipt of funds, and unreadable amount transfers. We explain Rule 1 later in this chapter.[3] Transactions use cryptography, so we call their content a *cryptocurrency*.

If two computers receive different 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.

Clearly 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.

[2] In political science this is called a Schelling Point [64], social minima, or social contract.
[3] In commodity money like gold these rules are met by physical reality.

7.1.2 Simple blockchain

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

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

$$BH_G = \mathcal{H}(\text{“Let the SampleCoin begin!”})$$

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

$$BH_1 = \mathcal{H}(BH_G, TH_1, TH_2, ...)$$
$$BH_2 = \mathcal{H}(BH_1, TH_3, TH_4, ...)$$

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/chains all the way back to the genesis message. We have a very simple 'blockchain'.[4]

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.

If different blocks referencing the same previous block are published at the same time, then the network of nodes will fork as each node receives one of the new blocks before the other (for simplicity, imagine about half the nodes end up with each side of the fork).

7.2 Difficulty

If nodes can publish new blocks whenever they want, the network might fracture and diverge into many different, equally legitimate, chains. Say it takes 30 seconds to make sure everyone in the network gets a new block. What if new blocks are sent out every 31, 15 seconds, 10 seconds, etc?

We can control 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 most nodes, the network will tend to remain intact.

7.2.1 Mining a block

The output of a cryptographic hash function is uniformly distributed and apparently independent of the input. This means, given a potential 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.

[4] 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]

Let's imagine a hash function $\mathcal{H}_i(x)$ which outputs a number from 1 to 100: $\mathcal{H}_i(x) \in_R^D \{1, ..., 100\}$.[5] 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 told to find a 'nonce' n (some integer) such that $\mathcal{H}_i(\mathfrak{m}, n)$ outputs a number less than or equal to the *target* $t = 5$ (i.e. $\mathcal{H}_i(\mathfrak{m}, n) \in \{1, ..., 5\}$).

Since only $1/20^{\text{th}}$ of outputs from $\mathcal{H}_i(x)$ will meet the target, it should take around 20 guesses of n to find one that works (and hence 20 minutes of computing time).

Searching for a useful nonce is called *mining*, and publishing the message with its nonce is a *proof of work* because it proves we found a useful nonce (even if we were lucky and found it with just one hash, or even blindly published a good nonce), 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), a nonce n to mine, and a target t, we can define the expected average number of hashes, the *difficulty* d, like this: $d = m/t$. If $\mathcal{H}_{PoW}(\mathfrak{m}, n) * d \leq m$, then $\mathcal{H}_{PoW}(\mathfrak{m}, n) \leq t$ and n is acceptable.[6]

src/cryptonote_basic/difficulty.cpp `check_hash()`

With smaller targets the difficulty rises and it takes a computer more and more hashes, and therefore longer and longer periods of time, to find useful nonces.[7]

7.2.2 Mining speed

Assume all nodes are mining 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 collect a bunch b of recent blocks from the blockchain (say, with index $u \in \{1, ..., b\}$) which each had a difficulty d_u. For now, assume the nodes who mined them were honest, so each block timestamp TS_u is accurate.[8] The total time between the earliest block and most recent block is $totalTime = TS_b - TS_1$. The approximate number of hashes it took to mine all the blocks is $totalDifficulty = \sum_u d_u$.

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[9]

$$hashSpeed \approx totalDifficulty/totalTime$$

[5] We use $\in_R^D$ to say the output is deterministically random.

[6] In Monero only difficulties are recorded/computed since $\mathcal{H}_{PoW}(\mathfrak{m}, n) * d \leq m$ doesn't need t.

[7] Mining and verifying are asymmetric since it takes the same time to verify a proof of work (one computation of the proof of work algorithm) no matter what the difficulty is.

[8] Timestamps are determined when a miner *starts* mining a block, so they are likely to lag behind the actual publication moment. The next block starts mining right away, so the timestamp that appears *after* a given block indicates how long miners spent on it.

[9] 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 for a given block of messages. As we will see, since miners include a miner transaction with one-time address $K^o \in_{ER} \mathbb{Z}_l$ (ER = effectively random) in their blocks, blocks are always unique between miners except with negligible probability, so trying the same nonces doesn't matter.

If we want to set the target time to mine new blocks so blocks are produced at a rate (one block)/(target time), then we calculate how many hashes it should take for the network to spend that amount of time mining. Note: we round up so the difficulty never equals zero.

$$newDifficulty = hashSpeed * targetTime$$

There is no guarantee the next block will take *newDifficulty* amount of total network 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*.[10]

7.2.3 Consensus: largest cumulative difficulty

Now we can resolve conflicts between chain forks.

By convention, the chain with highest cumulative difficulty (from all blocks in the chain), and therefore with most work (network hashes) spent constructing, is considered the real, legitimate version. If a chain splits and each fork has the same cumulative difficulty, nodes continue mining on the branch they received first. When one branch gets ahead of the other they discard ('orphan') the weaker branch.

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.[11]

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 (which meanwhile continues to grow). This is very hard to do unless you control over 50% of the network hash speed and can outwork other miners. [97]

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

[10] 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 weights, which we explore in Section 7.3.3.

[11] Monero developers have successfully changed its protocol 11 times, with nearly all users and miners adopting each fork: v1 April 18, 2014 (genesis version) [127]; v2 March 2016; v3 September 2016; v4 January 2017; v5 April 2017; v6 September 2017; v7 April 2018; v8 and v9 October 2018; v10 and v11 March 2019; v12 November 2019. The core git repository's README contains a summary of protocol changes in each version.

src/hardforks/hardforks.cpp `mainnet_hard_forks[]`

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.

Before chopping outliers we sorted timestamps, but *only* timestamps. Block difficulties are left unsorted. We use the cumulative difficulty for each block, which 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 sorted timestamps and unsorted 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 $o = 60$ (2 hours).[12,13]

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.

Rule 3: After w blocks, chop off high and low outliers, scaling the amount chopped until b blocks. If the amount of previous blocks (minus w) 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 has used a few different proof of work hash algorithms (with 32 byte outputs) in different protocol versions. The original, known as Cryptonight, was designed to be relatively inefficient on GPU, FPGA, and ASIC architectures [124] compared to standard hash functions like SHA256. In April 2018 (v7 of the protocol), new blocks were required to begin using a slightly modified variant that countered the advent of Cryptonight ASICs [29]. Another slight variant, named Cryptonight V2, was implemented in October 2018 (v8) [11], and Cryptonight-R (based on Cryptonight but with more substantial changes than just a tweak) started being used for new blocks in March 2019 (v10) [12]. A radical new proof of work called RandomX [70] was designed and made mandatory for new blocks in November 2019 (v12) with the intention of long-term ASIC resistance [15].

src/cryptonote_basic/cryptonote_tx_utils.cpp `get_block_longhash()`

src/crypto/rx-slow-hash.c

[12] In March 2016 (v2 of the protocol), Monero changed from 1 minute target block times to 2 minute target block times [14]. Other difficulty parameters have always been the same.

[13] Monero's difficulty algorithm may be suboptimal compared to state of the art algorithms [144]. Fortunately it is 'fairly resilient to selfish mining' [51], an essential feature.

7.3 Money supply

There are two basic mechanisms for creating money in a blockchain-based cryptocurrency.

First, the currency's creators can conjure coins and distribute them to people in the genesis message. This is often called an 'airdrop'. Sometimes creators give themselves a large amount in a so-called 'pre-mine'. [20]

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 supply increases indefinitely.

Monero is based on a currency known as Bytecoin that had a sizeable pre-mine, followed by block rewards [13]. 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 inflationary.

7.3.1 Block reward

Block miners, before mining for a nonce, make a 'miner transaction' with no inputs and at least one output.[14] The total 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.

src/cryptonote_core/blockchain.cpp `validate_miner_transaction()`

Besides distributing money, block rewards incentivize mining. If there were no block rewards (and no other mechanism), why would anyone mine new blocks? Perhaps altruism or curiosity. However, few miners makes it easy for a malicious actor to assemble >50% of the network's hash rate and easily rewrite recent chain history.[15] This is also why in Monero block rewards do not fall all the way to zero.

With block rewards, competition between miners drives total hash rate up until the marginal cost of adding more hash rate is higher than the marginal reward of obtaining that proportion of mined blocks (which appear at a constant rate) (plus some premiums like risk and opportunity cost). This means as a cryptocurrency becomes more valuable, its total hash rate will increase and it becomes progressively more difficult and expensive to gather >50%.

[14] A miner transaction can have any number of outputs, although currently the core implementation is only able to make one. Moreover, unlike normal transactions there are no explicit restrictions on miner transaction weight. They are functionally limited by the maximum block weight.

[15] As an attacker gets higher shares of the hash rate (beyond 50%), it takes less time to rewrite older and older blocks. Given a block x days old, owned hash speed v, and honest hash speed v_h ($v > v_h$), it will take $y = x * (v_h/(v - v_h))$ days to rewrite.

Bit shifting

Bit shifting is used for calculating the base block reward (as we will see in Section 7.3.3, the actual block reward can sometimes be reduced below the base amount).

Suppose we have an integer A = 13 with bit representation [1101]. If we shift the bits of A down by 2 using the bitwise shift right operator, denoted A >> 2, we get [0011].01, which equals 3.25. In reality that last part gets thrown away - 'shifted' into oblivion, leaving us with [0011] = 3.[16]

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).[17] In the beginning of Monero the base block reward was B = (L-M) >> 20. 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 Monero can't be divided. Clearly atomic units are ridiculous - L is over 18 quintillion! We can divide everything by 10^{12} 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 [127], was about 17.6 Moneroj! See Appendix C to confirm this for yourself.[18]

As blocks are mined M grows, lowering subsequent 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 [14]. To keep the rate of money creation, i.e. the 'emission schedule',[19] 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

src/cryptonote_basic/cryptonote_basic_impl.cpp `get_block_reward()`

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

7.3.2 Dynamic block weight

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

[16] Bitwise shift right by n bits is equivalent to integer division by 2^n.
[17] Perhaps now it is clear why range proofs (Section 5.5) limit transaction amounts to 64 bits.
[18] Monero amounts are stored in atomic-unit format in the blockchain.
[19] For an interesting comparison of Monero and Bitcoin's emission schedules see [19].

One mitigation is a fixed block size (in bytes), 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 focus on mining transactions with the highest fees. 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.[20]

Monero avoids those extremes (unlimited vs fixed) with a dynamic block weight.

Size vs Weight

Since Bulletproofs were added (v8), transaction and block sizes are no longer considered strictly. The term used now is *transaction weight*. Transaction weight for a miner transaction (see Section 7.3.6), or a normal transaction with two outputs, is equal to the size in bytes. When a normal transaction has more than two outputs the weight is somewhat higher than the size.

src/cryptonote_basic/cryptonote_format_utils.cpp `get_transaction_weight()`

Recalling Section 5.5, a Bulletproof occupies $(2 \cdot \lceil \log_2(64 \cdot p) \rceil + 9) \cdot 32$ bytes, so as more outputs are added the additional storage for range proofs is sub-linear. However, Bulletproof verification is linear, so artificially increasing transaction weights 'prices in' that extra verification time (it's called a 'clawback').

Suppose we have a transaction with p outputs, and imagine that if p isn't a power of 2 we create enough dummy outputs to fill the gap. We find the difference between the actual Bulletproof size, and the size of all the Bulletproofs if those p + 'dummy outputs' had been in 2-out transactions (it's 0 if $p = 2$). We only claw back 80% of the difference.[21]

src/cryptonote_basic/cryptonote_format_utils.cpp `get_transaction_weight_clawback()`

$$\text{transaction\_clawback} = 0.8 * [(23 * (p + \text{num\_dummy\_outs})/2) \cdot 32 - (2 \cdot \lceil \log_2(64 \cdot p) \rceil + 9) \cdot 32]$$

Therefore the transaction weight is

$$\text{transaction\_weight} = \text{transaction\_size} + \text{transaction\_clawback}$$

A block's weight is equal to the sum of its component transactions' weights plus the miner transaction's weight.

src/cryptonote_core/blockchain.cpp `create_block_template()`

Long term block weight

If dynamic blocks are allowed to grow at a rapid pace the blockchain can quickly become unmanageable [83]. To mitigate this, maximum block weights are tethered by *long term block weights*.

[20] Bitcoin has a history of overloaded transaction volume. This website (https://bitcoinfees.info/) charts the ridiculous fee levels encountered (up to the equivalent of 35$ per transaction at one point).

[21] Note that $\log_2(64 \cdot 2) = 7$, and $2 * 7 + 9 = 23$.

Each block has, in addition to its normal weight, a 'long term weight' calculated based on the previous block's effective median long term weight.[22] A block's effective median long term weight is related to the median of the most recent 100000 blocks' long term weights (including its own).[23,24]

src/cryptonote_core/blockchain.cpp `update_next_cumulative_weight_limit()`

$$\text{longterm\_block\_weight} = min\{\text{block\_weight}, 1.4 * \text{previous\_effective\_longterm\_median}\}$$
$$\text{effective\_longterm\_median} = max\{300\text{kB}, \text{median\_100000blocks\_longterm\_weights}\}$$

If normal block weights stay large for a long time, then it will take at least 50,000 blocks (about 69 days) for the effective long term median to rise by 40% (that's how long it takes a given long term weight to become the median).

Cumulative median weight

Transaction volume can change dramatically in a short period of time, especially around holidays [128]. To accommodate this, Monero allows short term flexibility in block weights. To smooth out transient variability, a block's cumulative median uses the median of the last 100 blocks' normal block weights (including its own).

`CRYPTONOTE_REWARD_BLOCKS_WINDOW`

$$\text{cumulative\_weights\_median} = max\{300\text{kB}, min\{max\{300\text{kB}, \text{median\_100blocks\_weights}\}, 50 * \text{effective\_longterm\_median}\}\}$$

The next block to be added to the blockchain is constrained in this way:[25]

$$\text{max\_next\_block\_weight} = 2 * \text{cumulative\_weights\_median}$$

src/cryptonote_basic/cryptonote_basic_impl.cpp `get_block_reward()`

While the maximum block weight can rise up to 100 times the effective median long term weight after a few hundred blocks, it cannot rise more than 40% beyond that over the next 50,000 blocks. Therefore long-term block weight growth is tethered by the long term weights, and in the short term weights may surge above their steady-state values.

7.3.3 Block reward penalty

To mine blocks bigger than the cumulative median, miners have to pay a price, or penalty, in the form of reduced block reward. This means there are functionally two zones within the maximum

[22] Similar to block difficulties, block weights and long term block weights are calculated and stored by blockchain verifiers rather than being included in blockchain data.

[23] Blocks made before long term weights were implemented have long term weights equal to their normal weights, so there is no concern for us about details surrounding the genesis block or early blocks. A brand new chain could easily make sensible choices.

[24] In the beginning of Monero the '300kB' term was 20kB, then increased to 60kB in March 2016, (v2 of the protocol) [14], and has been 300kB since April 2017 (v5 of the protocol) [1]. This non-zero 'floor' within the dynamic block weight medians helps transient transaction volume changes when the absolute volume is low, especially in the early stages of Monero adoption.

src/cryptonote_basic/cryptonote_basic_impl.cpp `get_min_block_weight()`

[25] The cumulative median replaced 'M100' (a similar median term) in protocol v8. Penalties and fees described in the first edition of this report [33] used M100.

block weight: the penalty-free zone, and the penalty zone. The median can slowly rise, allowing progressively larger blocks with no penalty.

If the intended block weight is greater than the cumulative median, then, given base block reward B, the block reward penalty is

$$P = B * ((\text{block\_weight}/\text{cumulative\_weights\_median}) - 1)^2$$

The actual block reward is therefore[26]

src/cryptonote_basic/cryptonote_basic_impl.cpp `get_block_reward()`

$$B^{\text{actual}} = B - P$$
$$B^{\text{actual}} = B * (1 - ((\text{block\_weight}/\text{cumulative\_weights\_median}) - 1)^2)$$

Using the ^2 operation means penalties are sub-proportional to block weight. A block weight 10% larger than the previous cumulative_weights_median has just a 1% penalty, 50% larger is 25% penalty, 90% larger is 81% penalty, and so on. [19]

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

7.3.4 Dynamic minimum fee

To prevent malicious actors from flooding the blockchain with transactions that could be used to pollute ring signatures, and generally bloat it unnecessarily, Monero has a minimum fee per byte of transaction data.[27] Originally this was 0.01 XMR/KiB (added early during protocol v1) [80], then it became 0.002 XMR/KiB in September 2016 (v3).[28]

src/cryptonote_core/blockchain.cpp `check_fee()`

In January 2017 (v4), a dynamic fee per KiB algorithm [46, 45, 44, 73] was added,[29] and then along with transaction weight reductions due to Bulletproofs (v8) it changed from per KiB to per byte. The most important feature of the algorithm is that it prevents minimum possible total fees

[26] Before confidential transactions (RingCT) were implemented (v4), all amounts were communicated in clear text and in some early protocol versions split into chunks (e.g. 1244 → 1000 + 200 + 40 + 4). To reduce miner tx size, the core implementation chopped off the lowest significant digits of block rewards (anything less than 0.0001 Moneroj; see `BASE_REWARD_CLAMP_THRESHOLD`) in v2-v3. The extra little bit was not lost, just made available for future block rewards. More generally, since v2 the block reward calculation here is just an upper limit on the real block reward that can be dispersed in a miner tx's outputs. Also of note, very early transactions' outputs with cleartext amounts *not* split into chunks can't be used in ring signatures in the current implementation, so to spend them they are migrated into chunked, 'mixable', outputs, which can then be spent in normal RingCT transactions by creating rings out of other chunks with the same amount. Exact modern protocol rules around these ancient pre-RingCT outputs are not clear.

src/cryptonote_core/blockchain.cpp `validate_miner_transaction()`

[27] This minimum is enforced by the node consensus protocol, not the blockchain protocol. Most nodes won't relay a transaction to other nodes if it has a fee below the minimum (at least in part so only transactions likely to be mined by someone are passed along [45]), but they *will* accept a new block containing that transaction. In particular, this means there is no need to maintain backward compatibility with fee algorithms.

src/cryptonote_core/tx_pool.cpp `add_tx()`

[28] The unit KiB (kibibyte, 1 KiB = 1024 bytes) is different from kB (kilobyte, 1 kB = 1000 bytes).

[29] The base fee was changed from 0.002 XMR/KiB to 0.0004 XMR/KiB in April 2017 (v5 of the protocol) [1]. The first edition of this report described the original dynamic fee algorithm [33].

from exceeding the block reward (even with small block rewards and large block weights), which is thought to cause instability [98, 47, 59].[30]

The fee algorithm

We base our fee algorithm around a reference transaction [73] of weight 3000 bytes (similar to a basic `RCTTypeBulletproof2` 2-input, 2-output transaction, which is usually about 2600 bytes)[31], and the fees it would take to offset the penalty when the median is at its minimum (the smallest penalty-free zone, 300kB) [45]. In other words, the penalty induced by a 303kB block weight.

Firstly, the fee F to balance the marginal penalty MP from adding a transaction with weight TW to a block with weight BW, is

$$\begin{aligned} \mathrm{F} = \mathrm{MP} = \mathrm{B} * ((&[\mathrm{BW} + \mathrm{TW}]/\mathrm{cumulative\_median} - 1)^2 - \\ &\mathrm{B} * ((\mathrm{BW}/\mathrm{cumulative\_median} - 1)^2 \end{aligned}$$

Defining the block weight factor $\mathrm{WF}_b = (\mathrm{BW}/\mathrm{cumulative\_median} - 1)$, and transaction weight factor $\mathrm{WF}_t = (\mathrm{TW}/\mathrm{cumulative\_median})$, lets us simplify

$$\mathrm{F} = \mathrm{B} * (2 * \mathrm{WF}_b * \mathrm{WF}_t + \mathrm{WF}_t^2)$$

Using a block weighing 300kB (with a cumulative median at the default 300kB) and our reference transaction with 3000 bytes,

$$\begin{aligned} \mathrm{F}_{\mathrm{ref}} &= \mathrm{B} * (2 * 0 * \mathrm{WF}_t + \mathrm{WF}_t^2) \\ \mathrm{F}_{\mathrm{ref}} &= \mathrm{B} * \mathrm{WF}_t^2 \\ \mathrm{F}_{\mathrm{ref}} &= \mathrm{B} * (\frac{\mathrm{TW}_{\mathrm{ref}}}{\mathrm{cumulative\_median}_{\mathrm{ref}}})^2 \end{aligned}$$

This fee is spread out over 1% of the penalty zone (3000 out of 300000). We can spread the same fee over 1% of any penalty zone with a generalized reference transaction.

$$\begin{aligned} \frac{\mathrm{TW}_{\mathrm{ref}}}{\mathrm{cumulative\_median}_{\mathrm{ref}}} &= \frac{\mathrm{TW}_{\text{general-ref}}}{\mathrm{cumulative\_median}_{\mathrm{general}}} \\ 1 &= (\frac{\mathrm{TW}_{\text{general-ref}}}{\mathrm{cumulative\_median}_{\mathrm{general}}}) * (\frac{\mathrm{cumulative\_median}_{\mathrm{ref}}}{\mathrm{TW}_{\mathrm{ref}}}) \\ \mathrm{F}_{\text{general-ref}} &= \mathrm{F}_{\mathrm{ref}} \\ &= \mathrm{F}_{\mathrm{ref}} * (\frac{\mathrm{TW}_{\text{general-ref}}}{\mathrm{cumulative\_median}_{\mathrm{general}}}) * (\frac{\mathrm{cumulative\_median}_{\mathrm{ref}}}{\mathrm{TW}_{\mathrm{ref}}}) \\ \mathrm{F}_{\text{general-ref}} &= \mathrm{B} * (\frac{\mathrm{TW}_{\text{general-ref}}}{\mathrm{cumulative\_median}_{\mathrm{general}}}) * (\frac{\mathrm{TW}_{\mathrm{ref}}}{\mathrm{cumulative\_median}_{\mathrm{ref}}}) \end{aligned}$$

[30] Credit for the concepts in this section largely belongs to Francisco Cabañas (a.k.a. 'ArticMine'), the architect of Monero's dynamic block and fee system. See [46, 45, 44].

[31] A basic 1-input, 2-output Bitcoin transaction is 250 bytes [26], or 430 bytes for 2-in/2-out.

Now we can scale the fee based on a real transaction weight at a given median, so e.g. if the transaction is 2% of the penalty zone the fee gets doubled.

$$\mathrm{F}_{\text{general}} = \mathrm{F}_{\text{general-ref}} * \frac{\mathrm{TW}_{\text{general}}}{\mathrm{TW}_{\text{general-ref}}}$$

$$\mathrm{F}_{\text{general}} = \mathrm{B} * (\frac{\mathrm{TW}_{\text{general}}}{\text{cumulative\_median}_{\text{general}}}) * (\frac{\mathrm{TW}_{\text{ref}}}{\text{cumulative\_median}_{\text{ref}}})$$

This rearranges to the default fee per byte, which we have been working toward.

$$f^{B}_{default} = \mathrm{F}_{\text{general}}/\mathrm{TW}_{\text{general}}$$

$$f^{B}_{default} = \mathrm{B} * (\frac{1}{\text{cumulative\_median}_{\text{general}}}) * (\frac{3000}{300000})$$

When transaction volume is below the median there is no real reason for fees to be at the reference level [73]. We set the minimum to be $1/5^{\text{th}}$ the default.

$$f^{B}_{min} = \mathrm{B} * (\frac{1}{\text{cumulative\_weights\_median}}) * (\frac{3000}{300000}) * (\frac{1}{5})$$

$$f^{B}_{min} = \mathrm{B} * (\frac{1}{\text{cumulative\_weights\_median}}) * 0.002$$

The fee median

It turns out using the cumulative median for fees enables a spam attack. By raising the short term median to its highest value (50 x long term median), an attacker can use minimum fees to maintain high block weights (relative to organic transaction volume) with very low cost.

To avoid this we limit fees for transactions to go in the next block with the smallest median available, which favors higher fees in all cases.[32]

$$\text{smallest\_median} = max\{300\text{kB}, min\{\text{median\_100blocks\_weights}, \text{effective\_longterm\_median}\}\}$$

src/cryptonote_core blockchain.cpp `check_fee()`

Favoring higher fees during rising transaction volume also facilitates adjusting the short term median and ensuring transactions aren't left pending, as miners are more likely to mine into the penalty zone.

[32] An attacker can spend just enough in fees for the short term median to hit 50*long-term-median. With current (as of this writing) block rewards at 2 XMR, an optimized attacker can increase the short term median by 17% every 50 blocks, and reach the upper bound after about 1300 blocks (about 43 hours), spending 0.39*2 XMR per block, for a total setup cost of about 1000 XMR (or around 65k USD at current valuations), and then go back to the minimum fee. When the fee median equals the penalty-free zone, then the minimum total fee to fill the penalty-free zone is 0.004 XMR (about 0.26 USD at current valuations). If the fee median equals the long term median, it would in the spam scenario be 1/50th the penalty-free zone. Therefore it would just be 50x the short-median case, for 0.2 XMR per block (13 USD per block). This comes out to 2.88 XMR per day vs 144 XMR per day (for 69 days, until the long term median rises by 40%) to maintain every block with 50*long-term-median block weight. The 1000 XMR setup cost would be worthwhile in the former case, but not the latter. This will reduce to 300 XMR setup, and 43 XMR maintenance, at the emission tail.

The actual minimum fee is therefore[33,34]

$$f^B_{min-actual} = \mathrm{B} * (\frac{1}{\text{smallest\_median}}) * 0.002$$

src/cryptonote_core blockchain.cpp `get_dynamic_base_fee()`

Transaction fees

As Cabañas said in his insightful presentation on this topic [45], "[f]ees tell the miner how deep into the penalty [transaction authors are] willing to pay for, in order to get a transaction mined." Miners will fill up their blocks by adding transactions in descending order of fee amount [45] (assuming all transactions have the same weight), so to move into the penalty zone there must be numerous transactions with large fees. This means it is likely the block weight cap can only be reached if total fees are at least about 3-4 times the base block reward (at which point the actual block reward is zero).[35]

To calculate fees for a transaction, Monero's core implementation wallet uses 'priority' multipliers. A 'slow' transaction uses the minimum fee directly, 'normal' is the default fee (5x), if all transactions use 'fast' (25x) they can reach 2.5% of the penalty zone, and a block with 'super urgent' (1000x) transactions can fill 100% of the penalty zone.

src/wallet/wallet2.cpp `get_fee_multiplier()`

One important consequence of dynamic block weights is average total block fees will tend to be of a magnitude lower than, or at least the same as, the block reward (total fees can be expected to equal the base block reward at about 37% of the penalty zone [68.5% of the maximum block weight], when the penalty is 13%). Transactions competing for block space with higher fees leads to a bigger supply of block space, and lower fees.[36] This feedback mechanism is a strong counter to the renowned 'selfish miner' [59] threat.

7.3.5 Emission tail

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

[33] To check if a given fee is correct, we allow a 2% buffer on $f^B_{min-actual}$ in case of integer overflow (we must compute fees before tx weights are completely determined). This means the effective minimum fee is $0.98 * f^B_{min-actual}$.

src/cryptonote_core blockchain.cpp `check_fee()`

[34] Research to improve minimum fees even further is ongoing. [131]

[35] The marginal penalty from the last bytes to fill up a block can be considered a 'transaction' comparable to other transactions. In order for a clump of transactions to buy that transaction space from a miner, all its individual transaction fees should be higher than the penalty, since if any one of them is lower then the miner will keep the marginal reward instead. This last marginal reward, assuming a block filled with small transactions, requires at least 4x the base block reward in total fees to be purchased. If transaction weights are maximized (50% of the minimum penalty-free zone, i.e. 150kB) then if the median is minimized (300kB) the last marginal transaction requires at least 3x in total fees.

[36] As block rewards decline over time, and the median rises due to increased adoption (theoretically), fees should steadily become smaller and smaller. In 'real purchasing power' terms, this may be less impactful on transaction costs if the value of Moneroj rises due to adoption and economic deflation.

Block weights stabilize around the average rate of transactions submitted to the network, and transaction authors have no compelling reason to use transaction fees above the minimum, which would be zero according to Section 7.3.4.

This introduces an unstable, insecure situation. Miners have little to no incentive to mine new blocks, leading to a fall in network hash rate as returns on investment decline. Block times remain the same as difficulties adjust, but the cost of performing a double-spend attack may become feasible.[37] If minimum fees are forced to be non-zero then the 'selfish miner' [59] threat becomes realistic [47].

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

src/cryptonote_basic/cryptonote_basic_impl.cpp `get_block_reward()`

$$0.6 > ((L - M) >> 19)/10^{12}$$
$$\mathrm{M} > \mathrm{L} - 0.6 * 2^{19} * 10^{12}$$
$$\mathrm{M}/10^{12} > \mathrm{L}/10^{12} - 0.6 * 2^{19}$$
$$\mathrm{M}/10^{12} > 18,132,171.273709551615$$

the Monero chain will enter a so-called 'emission tail', with constant 0.6 XMR (0.3 XMR/minute) block rewards forever after.[38] This corresponds with about 0.9% yearly inflation to begin with, steadily declining thereafter.

7.3.6 Miner transaction: `RCTTypeNull`

A block's miner has the right to claim ownership of the fees provided in its transactions, and to mint new money in the form of a block reward. The mechanism is a miner transaction (a.k.a. coinbase transaction), which is similar to a normal transaction.[39]

src/cryptonote_core/cryptonote_tx_utils.cpp `construct_miner_tx()`

The output amount(s) of a miner transaction must be no more than the sum of transaction fees and block reward, and are communicated in clear text.[40] In place of an input, the block's height is recorded (i.e. "I claim the block reward and fees for the n^{th} block").

Ownership of the miner output(s) is assigned to a standard one-time address[41], with a corresponding transaction public key stored in the extra field. The funds are locked, unspendable, until the

[37] The case of fixed supply and fixed block weight, as in Bitcoin, is also thought to be unstable. [47]

[38] The Monero emission tail's estimated arrival is May 2022 [17]. The money supply limit L will be reached in May 2024, but since coin emission will no longer depend on the supply it will have no effect. Based on Monero's range proof, it will be impossible to send more money than L in one output, even if someone manages to accumulate more than that (and assuming they have wallet software that can handle that much).

[39] Apparently, at one point miner transactions could be constructed using deprecated transaction format versions, and could include some normal transaction (RingCT) components. The issues were fixed in protocol v12 after this hackerone report was published: [8].

[40] In the current version miners may claim less than the calculated block reward. The leftovers are pushed back into the emission schedule for future miners.

[41] The miner transaction output can theoretically be sent to a subaddress and/or use multisig and/or an encoded payment ID. We don't know if any implementations have any of those features.

60^{th} block after it is published [21].[42]

Since RingCT was implemented in January 2017 (v4 of the protocol) [16], people downloading a new copy of the blockchain compute a commitment to the miner transaction (a.k.a. tx) amount a, as $C = 1G + aH$, and store it for referral. This allows block miners to spend their miner transaction outputs just like a normal transaction's outputs, putting them in MLSAG rings with other normal and miner tx outputs.

src/blockchain_db/blockchain_db.cpp `add_transaction()`

Blockchain verifiers store each post-RingCT block's miner tx amount commitment, for 32 bytes each.

7.4 Blockchain structure

Monero's blockchain style is simple.

It starts with a genesis message of some kind (in our case basically a miner transaction dispersing the first block reward), which constitutes the genesis block (see Appendix C). The next block contains a reference to the previous block, in the form of block ID.

src/cryptonote_core/cryptonote_tx_utils.cpp `generate_genesis_block()`

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 the miner transaction).[43]

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

src/cryptonote_basic/cryptonote_format_utils.cpp `get_block_hashing_blob()`

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.[44] The proof of work and block ID hash the same information, except use different hash functions. Blocks are mined by, while $(PoW_{output} * difficulty) > 2^{256} - 1$, repeatedly changing the nonce and recalculating

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

7.4.1 Transaction ID

Transaction IDs are similar to the message signed by input MLSAG signatures (Section 6.2.2), but include the MLSAG signatures too.

The following information is hashed:

src/cryptonote_basic/cryptonote_format_utils.cpp `calculate_block_hash()`

src/cryptonote_core/blockchain.cpp `get_block_longhash()`

src/cryptonote_core/blockchain.cpp `is_tx_spendtime_unlocked()`

[42] The miner tx can't be locked for more or less than 60 blocks. If it is published in the 10^{th} block, its unlock height is 70, and it may be spent in the 70^{th} block or later.

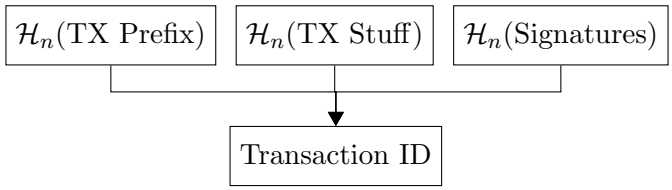
[43] +1 accounts for the miner tx.

[44] In Monero a typical miner (from https://monerobenchmarks.info/ as of this writing) can do less than 50,000 hashes per second, so less than 6 million hashes per block. This means the nonce variable doesn't need to be that big. Monero's nonce is 4 bytes (max 4.3 billion), and it would be strange for any miner to require all the bits.

- TX Prefix = {transaction era version (e.g. ringCT = 2), inputs {key offsets, key images}, outputs {one-time addresses}, extra {transaction public key, encoded payment ID, misc.}}
- TX Stuff = {signature type (`RCTTypeNull` or `RCTTypeBulletproof2`), transaction fee, pseudo output commitments for inputs, ecdhInfo (encrypted or cleartext amounts), output commitments}
- Signatures = {MLSAGs, range proofs}

src/cryptonote_basic/cryptonote_format_utils.cpp `calculate_transaction_hash()`

In this tree diagram the black arrow indicates a hash of inputs.



In place of an 'input', a miner transaction records the block height of its block. This ensures the miner transaction's ID, which is simply a normal transaction ID except with $\mathcal{H}_n$(Signatures) $\rightarrow$ 0, is always unique, for simpler ID-searching.

7.4.2 Merkle tree

Some users may want to discard data from their copy of the blockchain. For example, once you verify a 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 [92], which is just a binary hash tree. Any branch in a Merkle tree can be pruned if you keep its root hash.[45]

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.[46]

[45] The first known pruning method was added in v0.14.1 of the core Monero implementation (March 2019, coinciding with protocol v10). After verifying a transaction, full nodes can delete all its signature data (including Bulletproofs, MLSAGS, and pseudo output commitments) while keeping $\mathcal{H}_n$(Signatures) for computing the transaction ID. They only do this with $7/8^{\text{ths}}$ of all transactions, so every transaction is fully stored by at least $1/8^{\text{th}}$ of the network's full nodes. This reduces blockchain storage by about $2/3^{\text{rds}}$. [55]

[46] A bug in Monero's Merkle tree code led to a serious, though apparently non-critical, real-world attack on September 4^{th}, 2014 [86].

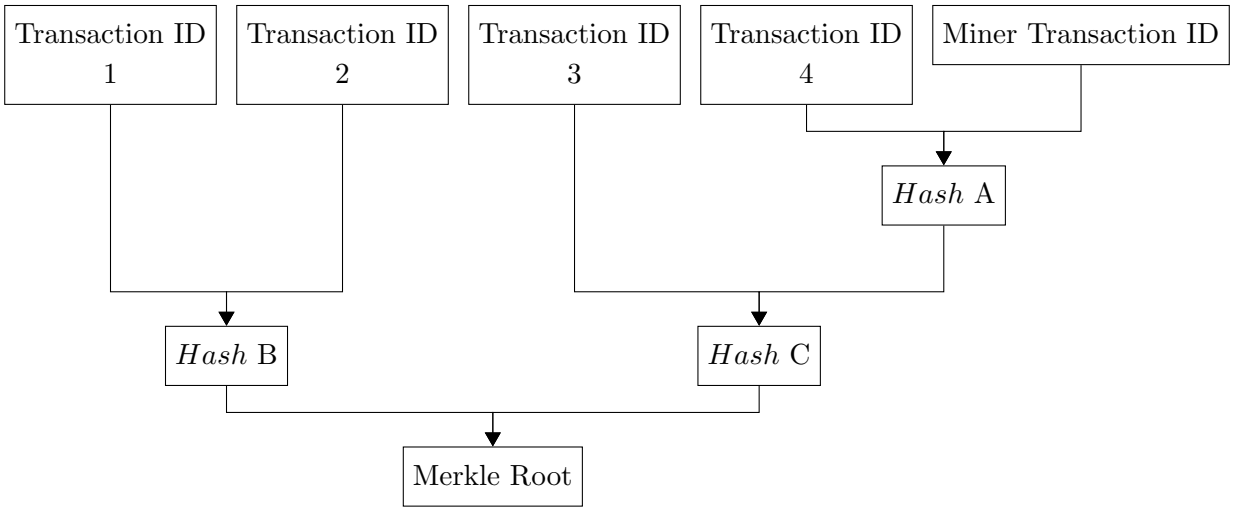


Figure 7.1: Merkle Tree

A Merkle root is inherently a reference to all its included transactions.

7.4.3 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 B.

- Block header:
  - **Major version**: Used to track hard forks (changes to the protocol).
  - **Minor version**: Once used for voting, now it just displays the major version again.
  - **Timestamp**: UTC (Coordinated Universal Time) time of the block. Added by miners, timestamps are unverified but they won't be accepted if lower than the median timestamp of the previous 60 blocks.
  - **Previous block's ID**: Referencing the previous block, this is the essential feature of a blockchain.
  - **Nonce**: A 4-byte integer that miners change over and over until the PoW hash meets the difficulty target. Block verifiers can easily recalculate the PoW hash.
- Miner transaction: Disperses the block reward and transaction fees to the block's miner.
- Transaction IDs: References to non-miner transactions 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()`

In addition to the data in each transaction (Section 6.3), we store the following information:

- Major and minor versions: variable integers ≤ 9 bytes
- Timestamp: variable integer ≤ 9 bytes
- Previous block's ID: 32 bytes
- Nonce: 4 bytes, can extend its effective size with the miner tx extra field's extra nonce[47]
- Miner transaction: 32 bytes for a one-time address, 32 bytes for a transaction public key (+1 byte for its 'extra' tag), and variable integers for the unlock time, corresponding block's height, and amount. After downloading the blockchain, we also need 32 bytes to store an amount commitment $C = 1G + aH$ (only for post-RingCT miner tx amounts).
- Transaction IDs: 32 bytes each

[47] Within each transaction is an 'extra' field which can contain more-or-less arbitrary data. If a miner needs a wider range of nonces than just 4 bytes, they can add or alter data in their miner tx's extra field to 'extend' the nonce size. [81]

Part II

Extensions

CHAPTER 8

Monero Transaction-Related Knowledge Proofs

Monero is a currency, and like any currency its uses are complex. From corporate accounting, to market exchange, to legal arbitration, different interested parties may want to know detailed information about transactions made.

How can you know for sure that money you received came from a specific person? Or prove that you did in fact send a certain output or transaction to someone despite claims to the contrary? Senders and recipients in the Monero public ledger are ambiguous. How can you prove you have a certain amount of money, without compromising your private keys? Amounts in Monero are completely hidden from observers.

We consider several types of transaction assertions, a few of which are implemented in Monero and available with built-in wallet tools. We also outline a framework for auditing the full balance owned by a person or organization, that doesn't require leaking information about future transactions they might make.

8.1 Transaction proofs in Monero

Monero transaction proofs are in the process of being updated [103]. The currently implemented proofs are all 'version 1', and don't include domain separation. We describe only the most advanced proofs, whether they be currently implemented, slated for implementation in future releases [104], or hypothetical proofs that may or may not get implemented (Sections 8.1.5 [130], and 8.2.1).

8.1.1 Multi-base Monero transaction proofs

There are a few details to be aware of going forward. Most Monero transaction proofs involve multi-base proofs (recall Section 3.1). Wherever relevant, the domain separator is $T_{txprf2} = \mathcal{H}_n(\text{“TXPROOF\_V2”})$.[1] The message being signed is usually (unless otherwise specified) $\mathfrak{m} = \mathcal{H}_n(\texttt{tx\_hash, message})$, where `tx_hash` is the relevant transaction's ID (Section 7.4.1), and `message` is an optional message that provers or third parties can provide to make sure the prover actually makes a proof and hasn't stolen it.

src/wallet/wallet2.cpp `get_tx_proof()`

Proofs are encoded in base-58, a binary-to-text encoding scheme first introduced for Bitcoin [28]. Verifying these proofs always involves first decoding them from base-58 back to binary. Note that verifiers also need access to the blockchain, so they can use transaction ID references to get information like one-time addresses.[2]

The structure of key prefixing in proofs is somewhat lopsided, due in part to accumulating updates that haven't reorganized it. Challenges for 2-base 'version 2' proofs are assembled with this format, where if 'base key 1' is G then its position in the challenge is filled with 32 zero bytes,

$c = \mathcal{H}_n(\mathfrak{m}$, public key 2, proof part 1, proof part 2, T_{txprf2}, public key 1, base key 2, base key 1)

8.1.2 Prove creation of a transaction input (`SpendProofV1`)

Suppose we made a transaction, and want to prove it. Clearly, by remaking a transaction input's signature on a new message, any verifier would have no choice but to conclude we made the original. Remaking *all* of a transaction's inputs' signatures means we must have made the entire transaction (recall Section 6.2.2), or at the very least fully funded it.[3]

A so-called 'SpendProof' contains remade signatures for all of a transaction's inputs. Importantly, SpendProof ring signatures re-use the original ring members to avoid identifying the true signer via ring intersections.

src/wallet/wallet2.cpp `get_spend_proof()`

SpendProofs are implemented in Monero, and to encode one for transmission to verifiers, the prover concatenates the prefix string “`SpendProofV1`” with the list of signatures. Note that the prefix string is not in base-58 and doesn't need to be encoded/decoded, since its purpose is human readability.

The SpendProof

SpendProofs unexpectedly don't use MLSAGs, but rather Monero's original ring signature scheme that was used in the very first transaction protocol (pre-RingCT) [136].

src/crypto/crypto.cpp `generate_ring_signature()`

[1] Just like in Section 3.6, hash functions should be domain separated by prefixing them with tags. The current Monero transaction proofs implementation has no domain separation, so all the tags in this chapter are in features *not* yet implemented.

[2] Transaction IDs are usually communicated separately from proofs.

[3] As we will see in Chapter 11, someone who made one input signature didn't necessarily make all input signatures.

1. Calculate key image $\tilde{K} = k_\pi^o \mathcal{H}_p(K_\pi^o)$.
2. Generate random number $\alpha \in_R \mathbb{Z}_l$ and random numbers $c_i, r_i \in_R \mathbb{Z}_l$ for $i \in \{1, 2, ..., n\}$ but excluding $i = \pi$.
3. Compute
$$c_{tot} = \mathcal{H}_n(\mathfrak{m}, [r_1 G + c_1 K_1^o], [r_1 \mathcal{H}_p(K_1^o) + c_1 \tilde{K}], ..., [\alpha G], [\alpha \mathcal{H}_p(K_\pi^o)], ..., \text{etc.})$$
4. Define the real challenge
$$c_\pi = c_{tot} - \sum_{i=1, i\neq\pi}^{n} c_i$$
5. Define $r_\pi = \alpha - c_\pi * k_\pi^o \pmod{l}$.

The signature is $\sigma = (c_1, r_1, c_2, r_2, ..., c_n, r_n)$.

Verification

To verify a SpendProof on a given transaction, the verifier confirms that all ring signatures are valid using information found in the relevant reference transaction (e.g. key images, and output offsets for getting one-time addresses from other transactions).

src/wallet/wallet2.cpp `check_spend_proof()`

1. Compute
$$c_{tot} = \mathcal{H}_n(\mathfrak{m}, [r_1 G + c_1 K_1^o], [r_1 \mathcal{H}_p(K_1^o) + c_1 \tilde{K}], ..., [r_n G + c_n K_n^o], [r_n \mathcal{H}_p(K_n^o) + c_n \tilde{K}])$$
2. Check that
$$c_{tot} \stackrel{?}{=} \sum_{i=1}^{n} c_i$$

Why it works

Note how this scheme is the same as bLSAG (Section 3.4) when there is only one ring member. To add a fake member, instead of passing the challenge $c_{\pi+1}$ into a new challenge hash, the member gets added into the original hash. Since the following equation
$$c_s = c_{tot} - \sum_{i=1, i\neq s}^{n} c_i$$
trivially holds for any index s, a verifier will have no way to identify the real challenge. Moreover, without knowledge of k_π^o the prover would never have been able to define r_π properly (except with negligible probability).

8.1.3 Prove creation of a transaction output (`OutProofV2`)

Now suppose we sent someone money (an output) and want to prove it. Transaction outputs contain at heart three components: the recipient's address, the amount sent, and the transaction private key. Amounts are encoded, so we only really need the address and transaction private key to get started. Anyone who deletes or loses their transaction private key will be unable to make an OutProof, so in that sense OutProofs are the least reliable of all Monero transaction proofs.[4]

Our task here is to show the one-time address was made from the recipient's address, and allow verifiers to reconstruct the output commitment. We do so by providing the sender-receiver shared secret rK^v, then proving we created it and that it corresponds with the transaction public key and recipient's address by signing a 2-base signature (Section 3.1) on the base keys G and K^v. Verifiers can use the shared secret to check the recipient (Section 4.2), decode the amount (Section 5.3), and reconstruct the output commitment (Section 5.3). We provide details for both normal addresses and subaddresses.

src/wallet/wallet2.cpp `check_tx_proof()`

The OutProof

To generate a proof for an output directed to an address (K^v, K^s) or subaddress $(K^{v,i}, K^{s,i})$, with transaction private key r, where the sender-receiver shared secret is rK^v, recall that the transaction public key stored in transaction data is either rG or $rK^{s,i}$ depending on whether or not the recipient is a subaddress (Section 4.3).

src/crypto/crypto.cpp `generate_tx_proof()`

1. Generate random number $\alpha \in_R \mathbb{Z}_l$, and compute
   (a) *Normal address*: αG and αK^v
   (b) *Subaddress*: $\alpha K^{s,i}$ and $\alpha K^{v,i}$
2. Calculate the challenge
   (a) *Normal address*:[5]
   $$c = \mathcal{H}_n(\mathfrak{m}, [rK^v], [\alpha G], [\alpha K^v], [T_{txprf2}], [rG], [K^v], [0])$$
   (b) *Subaddress*:
   $$c = \mathcal{H}_n(\mathfrak{m}, [rK^{v,i}], [\alpha K^{s,i}], [\alpha K^{v,i}], [T_{txprf2}], [rK^{s,i}], [K^{v,i}], [K^{s,i}])$$
3. Define the response[6] $r^{resp} = \alpha - c * r$.
4. The signature is $\sigma^{outproof} = (c, r^{resp})$.

[4] We can think of an 'OutProof' as showing an output is 'outgoing' from the prover. The corresponding 'InProofs' (Section 8.1.4) show outputs that are 'incoming' to the prover's address.

[5] Here the '0' value is a 32-byte encoding of zero bytes.

[6] Due to the limited number of available symbols, we unfortunately used r for both responses and the transaction private key. Superscript 'resp' for 'response' will be used to differentiate the two when necessary.

A prover can generate a bunch of OutProofs, and send them all together to a verifier. He concatenates prefix string "`OutProofV2`" with a list of proofs, where each item (encoded in base-58) consists of the sender-receiver shared secret rK^v (or $rK^{v,i}$ for a subaddress), and its corresponding $\sigma^{outproof}$. We assume the verifier knows the appropriate address for each proof.

src/wallet/wallet2.cpp `get_tx_proof()`

Verification

1. Calculate the challenge

   (a) *Normal address*:

   $$c' = \mathcal{H}_n(\mathfrak{m}, [rK^v], [r^{resp}G + c * rG], [r^{resp}K^v + c * rK^v], [T_{txprf2}], [rG], [K^v], [0])$$

   (b) *Subaddress*:

   $$c' = \mathcal{H}_n(\mathfrak{m}, [rK^{v,i}], [r^{resp}K^{s,i} + c*rK^{s,i}], [r^{resp}K^{v,i} + c*rK^{v,i}], [T_{txprf2}], [rK^{s,i}], [K^{v,i}], [K^{s,i}])$$

   src/crypto/crypto.cpp `check_tx_proof()`

2. If $c = c'$ then the prover knows r, and rK^v is legitimately a shared secret between rG and K^v (except with negligible probability).

3. The verifier should check the recipient's address provided can be used to make a one-time address from the relevant transaction (it's the same computation for normal addresses and subaddresses)

   $$K^s \stackrel{?}{=} K_t^o - \mathcal{H}_n(rK^v, t)$$

   src/wallet/wallet2.cpp `check_tx_key_helper()`

4. They should also decode the output amount b_t, compute the output mask y_t, and try to reconstruct the corresponding output commitment[7]

   $$C_t^b \stackrel{?}{=} y_t G + b_t H$$

8.1.4 Prove ownership of an output (`InProofV2`)

An OutProof shows the prover sent an output to an address, while an InProof shows an output was received to a certain address. It is essentially the other 'side' of the sender-receiver shared secret rK^v. This time the prover proves knowledge of k^v in K^v, and that in combination with the transaction public key rG the shared secret $k^v * rG$ appears.

Once a verifier has rK^v, they can check if the corresponding one-time address is owned by the prover's address with $K^o - \mathcal{H}_n(k^v * rG, t) * G \stackrel{?}{=} K^s$ (Section 4.2.1). By making an InProof for all transaction public keys on the blockchain, a prover will reveal all his owned outputs.

src/wallet/wallet2.cpp `check_tx_proof()`

[7] A valid OutProof signature doesn't necessarily mean the recipient considered is the real recipient. A malicious prover could generate a random view key K'^v, compute $K'^s = K^o - \mathcal{H}_n(rK'^v, t) * G$, and provide (K'^v, K'^s) as the nominal recipient. By recalculating the output commitment, verifiers can be more confident the recipient address in question is legitimate. However, a prover and recipient could collaborate to encode the output commitment using K'^v, while the one-time address uses (K^v, K^s). Since the recipient would need to know the private key k'^v (assuming the output amount is still meant to be spendable), there is questionable utility to that level of deception. Why wouldn't the recipient just use (K'^v, K'^s) (or some other single-use address) for the entire output? Since the computation of C_t^b is related to the recipient, we consider the described OutProof verification process adequate. In other words, the prover can't use it to deceive verifiers without coordinating with the recipient.

Giving the view key directly to a verifier would have the same effect, but once they have that key the verifier would be able to identify ownership of outputs to be created in the future. With InProofs the prover is able to retain control of his private keys, at the cost of the time it takes to prove (and then verify) each output is owned or unowned.

The InProof

An InProof is constructed the same way as an OutProof, except the base keys are now $\mathcal{J} = \{G, rG\}$, the public keys are $\mathcal{K} = \{K^v, rK^v\}$, and the signing key is k^v instead of r. We will show just the verification step to clarify our meaning. Note that the order of key prefixing changes (rG and K^v swap places) to coincide with the different role each key has.

src/crypto/crypto.cpp `generate_tx_proof()`

A multitude of InProofs, related to many outputs owned by the same address, can be sent together to the verifier. They are prefixed with the string "`InProofV2`", and each item (encoded in base-58) contains the sender-receiver shared secret rK^v (or $rK^{v,i}$), and its corresponding $\sigma^{inproof}$.

src/wallet/wallet2.cpp `get_tx_proof()`

Verification

1. Calculate the challenge

   src/crypto/crypto.cpp `check_tx_proof()`

   (a) *Normal address*:

   $$c' = \mathcal{H}_n(\mathfrak{m}, [rK^v], [r^{resp}G + c * K^v], [r^{resp} * rG + c * k^v * rG], [T_{txprf2}], [K^v], [rG], [0])$$

   (b) *Subaddress*:

   $$c' = \mathcal{H}_n(\mathfrak{m}, [rK^{v,i}], [r^{resp}K^{s,i} + c * K^{v,i}], [r^{resp} * rK^{s,i} + c * k^v * rK^{s,i}], [T_{txprf2}], [K^{v,i}], [rK^{s,i}], [K^{s,i}])$$

2. If $c = c'$ then the prover knows k^v, and $k^v * rG$ is legitimately a shared secret between K^v and rG (except with negligible probability).

Prove 'full' ownership with the one-time address key

While an InProof shows a one-time address was constructed with a specific address (except with negligible probability), it doesn't necessarily mean the prover can *spend* that output. Only those who can spend an output actually own it.

Proving ownership, once an InProof is complete, is as simple as signing a message with the spend key.[8]

[8] The ability to provide such a signature directly does not seem to be available in Monero, although as we will see ReserveProofs (Section 8.1.6) do include them.

8.1.5 Prove an owned output was not spent in a transaction (UnspentProof)

It would seem like proving an output is spent or unspent is as simple as recreating its key image with a multi-base proof on $\mathcal{J} = \{G, \mathcal{H}_p(K^o)\}$ and $\mathcal{K} = \{K^o, \tilde{K}\}$. While this does obviously work, verifiers must learn the key image, which also reveals when an unspent output is spent *in the future*.

It turns out we can prove an output wasn't spent in a specific transaction without revealing the key image. Moreover, we can prove it is currently unspent *full stop*, by extending this UnspentProof [130] to 'all the transactions where it was included as a ring member'.[9]

More specifically, our UnspentProof says that a given key image from a transaction on the blockchain does, or does not, correspond with a specific one-time address from its corresponding ring. Incidentally, as we will see, UnspentProofs go hand-in-hand with InProofs.

Setting up an UnspentProof

The verifier of an UnspentProof must know rK^v, the sender-receiver shared secret for a given owned output with one-time address K^o and transaction public key rG. He either knows the view key k^v, which allowed him to calculate $k^v * rG$ and check $K^o - \mathcal{H}_n(k^v * rG, t) * G \stackrel{?}{=} K^s$ so he knows the output being tested belongs to the prover (recall Section 4.2), or the prover provided rK^v. This is where InProofs come in, since with an InProof the verifier can be assured rK^v legitimately came from the prover's view key, and corresponds with an owned output, without learning the private view key.

Before verifying an UnspentProof, the verifier will learn the key image to be tested $\tilde{K}_?$, and checks that its corresponding ring includes the prover's owned output's one-time address K^o. He then calculates the partial 'spend' image $\tilde{K}^s_?$.

$$\tilde{K}^s_? = \tilde{K}_? - \mathcal{H}_n(rK^v, t) * \mathcal{H}_p(K^o)$$

If the tested key image was created from K^o then the resultant point will be $\tilde{K}^s_? = k^s * \mathcal{H}_p(K^o)$.

The UnspentProof

Our prover creates two multi-base proofs (recall Section 3.1). His address, which owns the output in question, is (K^v, K^s) or $(K^{v,i}, K^{s,i})$.[10]

1. A 3-base proof, where the signing key is k^s, and

$$\mathcal{J}_3^{unspent} = \{[G], [K^s], [\tilde{K}^s_?]\}$$
$$\mathcal{K}_3^{unspent} = \{[K^s], [k^s * K^s], [k^s * \tilde{K}^s_?]\}$$

[9] UnspentProofs have not been implemented in Monero.

[10] UnspentProofs are made the same way for subaddresses and normal addresses. The full spend key of a subaddress is required, e.g. $k^{s,i} = k^s + \mathcal{H}_n(k^v, i)$ (Section 4.3).

2. A 2-base proof, where the signing key is $k^s * k^s$, and

$$\mathcal{J}_2^{unspent} = \{[G], [\mathcal{H}_p(K^o)]\}$$
$$\mathcal{K}_2^{unspent} = \{[k^s * K^s], [k^s * k^s * \mathcal{H}_p(K^o)]\}$$

Along with proofs $\sigma_3^{unspent}$ and $\sigma_2^{unspent}$, the prover makes sure to communicate the public keys $k^s * K^s$, $k^s * \tilde{K}_?^s$, and $k^s * k^s * \mathcal{H}_p(K^o)$.

Verification

1. Confirm $\sigma_3^{unspent}$ and $\sigma_2^{unspent}$ are legitimate.
2. Make sure the same public key $k^s * K^s$ was used in both proofs.
3. Check whether $k^s * \tilde{K}_?^s$ and $k^s * k^s * \mathcal{H}_p(K^o)$ are the same. If they are, the output is spent, and if not it is unspent (except with negligible probability).

Why it works

This seemingly roundabout approach prevents the verifier from learning $k^s * H_p(K^o)$ for an unspent output, which he could use in combination with rK^v to compute its real key image, while leaving him confident the tested key image doesn't correspond to that output.

Proof $\sigma_2^{unspent}$ can be reused for any number of UnspentProofs involving the same output, although if it actually was spent then only one is really necessary (i.e. UnspentProofs can also be used to demonstrate an output is spent). Performing UnspentProofs on all ring signatures where a given unspent output was referenced should not be computationally expensive. An output is only likely, over time, to be included as decoys in on the order of 11 (current ring size) different rings.

8.1.6 Prove an address has a minimum unspent balance (`ReserveProofV2`)

Despite the privacy leak of revealing an output's key image when it isn't spent yet, it's still a somewhat useful method and was implemented in Monero [126] before UnspentProofs were invented [130]. Monero's so-called 'ReserveProof' is used to prove an address owns a minimum amount of money by creating key images for some unspent outputs.

src/wallet/wallet2.cpp `get_reserve_proof()`

More specifically, given a minimum balance, the prover finds enough unspent outputs to cover it, demonstrates ownership with InProofs, makes key images for them and proves they are legitimately based on those outputs with 2-base proofs (using a different key prefixing format), and then proves knowledge of the private spend keys used with normal Schnorr signatures (there may be more than one if some outputs are owned by different subaddresses). A verifier can check that the key images have not appeared on the blockchain, and hence their outputs must be unspent.

The ReserveProof

All the sub-proofs within a ReserveProof sign a different message than other proofs (e.g. OutProofs, InProofs, or SpendProofs). This time it is $\mathfrak{m} = \mathcal{H}_n(\texttt{message}, \texttt{address}, \tilde{K}_1^o, ..., \tilde{K}_n^o)$, where `address` is the encoded form (see [5]) of the prover's normal address (K^v, K^s), and the key images correspond with unspent outputs to be included in the proof.

1. Each output has an InProof, which shows the prover's address (or one of his subaddresses) owns the output.
2. Each output's key image is signed with a 2-base proof, where the challenge is formatted like this
$$c = \mathcal{H}_n(\mathfrak{m}, [rG + c * K^o], [r\mathcal{H}_p(K^o) + c * \tilde{K}])$$
src/crypto/crypto.cpp `generate_ring_signature()`
3. Each address (and subaddress) that owns at least one output has a normal Schnorr signature (Section 2.3.5), and the challenge looks like (it's the same for normal addresses and subaddresses)
$$c = \mathcal{H}_n(\mathfrak{m}, K^{s,i}, [rG + c * K^{s,i}])$$
src/crypto/crypto.cpp `generate_signature()`

To send a ReserveProof to someone else, the prover concatenates prefix string "`ReserveProofV2`" with two lists encoded in base-58 (e.g. "`ReserveProofV2`, list 1, list 2"). Each item in list 1 is related to a specific output and contains its transaction hash (Section 7.4.1), output index in that transaction (Section 4.2.1), the relevant shared secret rK^v, its key image, its InProof $\sigma^{inproof}$, and its key image proof. List 2 items are the addresses that own those outputs along with their Schnorr signatures.
src/wallet/wallet2.cpp `get_reserve_proof()`

Verification

1. Check the ReserveProof key images have not appeared in the blockchain.
2. Verify the InProof for each output, and that one of the provided addresses owns each one.
3. Verify the 2-base key image signatures.
4. Use the sender-receiver shared secrets to decode the output amounts (Section 5.3).
5. Check each address's signature.

src/wallet/wallet2.cpp `check_reserve_proof()`

If everything is legitimate, then the prover must own, unspent, at least the total amount contained in the ReserveProof's outputs (except with negligible probability).[11]

[11] ReserveProofs, while demonstrating full ownership of funds, do not include proofs that given subaddresses actually correspond with the prover's normal address.

8.2 Monero audit framework

In the USA most companies undergo yearly audits of their financial statements [129], which include the income statement, balance sheet, and cash flow statement. Of these the former two involve in large part a company's internal record-keeping, while the last involves every transaction that affects how much money the company currently has. Cryptocurrencies are digital cash, so any audit of a cryptocurrency user's cash flow statement must relate to transactions stored on the blockchain.

The first task of an audited person is to identify all the outputs they currently own (spent and unspent). This can be one with InProofs using all of their addresses. A large business may have a multitude of subaddresses, especially retailers operating in online marketplaces (see Chapter 10). Creating InProofs on all transactions for every single subaddress may result in enormous computational and storage requirements for both provers and verifiers.

Instead, we can make InProofs for just the prover's normal addresses (on all transactions). The auditor uses those sender-receiver shared secrets to check if any outputs are owned by the prover's main address or its related subaddresses. Recalling Section 4.3, a user's view key is enough to identify all outputs owned by an address's subaddresses.

To ensure the prover is not hoodwinking an auditor by hiding the normal address for some of his subaddresses, he also must prove all subaddresses correspond with one of his known normal addresses.

8.2.1 Prove an address and subaddress correspond (SubaddressProof)

SubaddressProofs show that a normal address's view key can be used to identify outputs owned by a given subaddress.[12]

The SubaddressProof

SubaddressProofs can be made in much the same way as OutProofs and InProofs. Here the base keys are $\mathcal{J} = \{G, K^{s,i}\}$, public keys are $\mathcal{K} = \{K^v, K^{v,i}\}$, and signing key is k^v. Again, we show just the verification step to clarify our meaning.

Verification

A verifier knows the prover's address (K^v, K^s), subaddress $(K^{v,i}, K^{s,i})$, and has the SubaddressProof $\sigma^{subproof} = (c, r)$.

[12] SubaddressProofs have not been implemented in Monero.

1. Calculate the challenge
$$c' = \mathcal{H}_n(\mathfrak{m}, [K^{v,i}], [rG + c * K^v], [rK^{s,i} + c * K^{v,i}], [T_{txprf2}], [K^v], [K^{s,i}], [0])$$
2. If $c = c'$ then the prover knows k^v for K^v, and $K^{s,i}$ in combination with that view key makes $K^{v,i}$ (except with negligible probability).

8.2.2 The audit framework

Now we are prepared to learn as much as possible about a person's transaction history.[13]

1. The prover gathers a list of all his accounts, where each account consists of a normal address and various subaddresses. He makes SubaddressProofs for all subaddresses. Much like ReserveProofs, he also makes a signature with the spend key of each address and subaddress, demonstrating he has spend-rights over all outputs owned by those addresses.
2. The prover generates, for each of his normal addresses, InProofs on all transactions (e.g. all transaction public keys) in the blockchain. This reveals to the auditor all outputs owned by the prover's addresses since they can check all one-time addresses with the sender-receiver shared secrets. They can be sure outputs owned by subaddresses will be identified, because of the SubaddressProofs.[14]
3. The prover generates, for each of his owned outputs, UnspentProofs on all transaction inputs where they appear as ring members. Now the auditor will know the prover's balance, and can further investigate spent outputs.[15]
4. *Optional*: The prover generates, for each transaction where he spent an output, an OutProof to show the auditor the recipient and amount. This step is only possible for transactions where the prover saved the transaction private key(s).

Importantly, a prover has no way to show the origin of funds directly. His only recourse is to request a set of proofs from people sending him money.

1. For a transaction sending money to the prover, its author makes a SpendProof demonstrating they actually sent it.
2. The prover's funder also makes a signature with an identifying public key, for example the spend key of their normal address. Both the SpendProof and this signature should sign a message containing that identifying public key, to ensure the SpendProof wasn't stolen or in fact made by someone else.

[13] This audit framework is not completely available in Monero. SubaddressProofs and UnspentProofs are not implemented, InProofs are not prepared for the optimization related to subaddresses that we explained, and there is no real structure to easily get or organize all the necessary information for both provers and verifiers.

[14] This step can also be completed by providing the private view keys, although it has obvious privacy implications.

[15] Alternatively, he could make ReserveProofs for all owned outputs. Again, revealing the key images of unspent outputs has obvious privacy implications.

CHAPTER 9

Multisignatures in Monero

Cryptocurrency transactions are not reversible. If someone steals private keys or succeeds in a scam, the money lost could be gone forever. Dividing signing power between people can weaken the potential danger of a miscreant.

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 collaborative signing with so-called 'M-of-N multisig'. In M-of-N, N people cooperate to make a joint key, and only M people ($M \leq N$) 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.

In this chapter we focus on how we feel multisig *should* be done, based on the recommendations in [111], and various observations about efficient implementation. We try to point out in footnotes where the current implementation deviates from what is described.[2] Our contributions are detailing M-of-N multisig, and a novel approach to nesting multisig keys.

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

[2] As of this writing we are aware of three multisig implementations. First is a very basic manual process using the CLI (command line interface) [114]. Second is the truly excellent MMS (Multisig Messaging System) which enables

9.1 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 observers, co-signers need to encrypt the messages they send each other.

Diffie-Hellman exchange (ECDH) is a very simple way to encrypt messages using elliptic curve cryptography. We already mentioned this in Section 5.3, where Monero output amounts are communicated to recipients via the shared secret rK^v. It looked like this:

$$amount_t = b_t \oplus_8 \mathcal{H}_n(\text{“}amount\text{”}, \mathcal{H}_n(rK_B^v, t))$$

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 along with the public key rG to the intended recipient, who can then decrypt the message with the shared secret krG. Message senders should also create a signature on their encrypted message (or just the encrypted message's hash for simplicity) so receivers can confirm messages weren't tampered with (a signature is only verifiable on the correct message $\mathtt{m}$).

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 [4], or see a technical description of the popular AES encryption scheme here [23]. Also, Dr. Bernstein developed an encryption scheme known as ChaCha [37, 100], which the primary Monero implementation uses to encrypt certain sensitive information related to users' wallets (such as key images for owned outputs).

src/wallet/wallet2.cpp `export_multisig()`
src/wallet/ringdb.cpp

9.2 Key aggregation for addresses

9.2.1 Naive approach

Let's say N people want to create a group multisignature address, which we denote $(K^{v,grp}, K^{s,grp})$. 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 group address, we can let everyone know the group view key $k^{v,grp}$ (recall Sections 4.1 and 4.2). To give all participants equal power, the view key can be a sum of view key components that all participants send each

secure, highly automated multisig via the CLI [115, 116]. Third is the commercially available 'Exa Wallet', which has initial release code available on their Github repository at https://github.com/exantech (it does not appear up to date with current release version). All three of these rely on the same fundamental core team's codebase, which essentially means only one implementation exists.

other securely. For participant $e \in \{1, ..., N\}$, his base view key component is $k_e^{v,base} \in_R \mathbb{Z}_l$, and all participants can compute the group private view key

$$k^{v,grp} = \sum_{e=1}^{N} k_e^{v,base}$$

In a similar fashion, the group spend key $K^{s,grp} = k^{s,grp}G$ could be a sum of private spend key base 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 and he can sign transactions on his own. It wouldn't be multisignature, just a plain old signature.

src/multisig/multisig.cpp `generate_multisig_view_secret_key()`

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

$$K^{s,grp} = \sum_{e} K_e^{s,base}$$

Clearly this is the same as

$$K^{s,grp} = (\sum_{e} k_e^{s,base}) * G$$

src/multisig/multisig.cpp `generate_multisig_N_N()`

9.2.2 Drawbacks to the naive approach

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

Key aggregation test

An outside adversary who knows all the base public spend keys $K_e^{s,base}$ can trivially test a given public address (K^v, K^s) for key aggregation by computing $K^{s,grp} = \sum_e K_e^{s,base}$ and checking $K^s \stackrel{?}{=} K^{s,grp}$. 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.[3]

We can get around this by creating new base 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,[4] let his new base private key components for the group address

[3] If at least one honest participant uses components selected randomly from a uniform distribution, then keys aggregated by a simple sum are indistinguishable [122] from normal keys.

[4] The random masks could easily be derived from some password. For example, $\mu^s = \mathcal{H}_n(password)$ and $\mu^v = \mathcal{H}_n(\mu^s)$. Or, as is done in Monero, mask the spend and view keys with a string e.g. $\mu^s, \mu^v =$ "Multisig". This implies Monero only supports one multisig base spend key per normal address, although in reality making a wallet multisig causes users to lose access to the original wallet [114]. Users must make a new wallet with their normal address to access its funds, assuming the multisig wasn't made from a brand new normal address.

src/multisig/multisig.cpp `get_multisig_blinded_secret_key()`

be

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

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.[5]

If key aggregation tests are not a concern, they could publish their public key base components $(K_e^{v,base}, K_e^{s,base})$ as normal addresses. Any third party could then compute the group address from those individual addresses and send funds to it, without interacting with any of the joint recipients [90].

Key cancellation

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

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

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

This leaves a group address $(k^{v,grp}G, K_B^{s,base})$ where Alice knows the private group 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, to make a signature proving they know the private key to their spend key component [109].[6] This is inconvenient and vulnerable to implementation mistakes. Fortunately a solid alternative is available.

9.2.3 Robust key aggregation

To easily resist key cancellation we make a small change to spend key aggregation (leaving view key aggregation the same). Let the set of N signers' base spend key components be $\mathbb{S}^{base} =$

[5] As we will see in Section 9.6, key aggregation does not work on M-of-N multisig when M < N due to the presence of shared secrets.

[6] Monero's current (and first) iteration of multisig, made available in April 2018 [62] (with M-of-N integration following in October 2018 [11]), used this naive key aggregation, and required users sign their spend key components.

src/wallet/wallet2.cpp `get_multisig_info()`

$\{K_1^{s,base}, ..., K_N^{s,base}\}$, ordered according to some convention (such as smallest to largest numerically, i.e. lexicographically).[7] The robust aggregated spend key is [111][8,9]

$$K^{s,grp} = \sum_e \mathcal{H}_n(T_{agg}, \mathbb{S}^{base}, K_e^{s,base}) K_e^{s,base}$$

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

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

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

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

Since participants don't need to prove they know their private spend keys, or really interact at all before 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"[90].[10]

Functions `premerge` and `merge`

More formally, and for the sake of clarity going forward, we can say there is an operation `premerge` which takes in a set of base keys $\mathbb{S}^{base}$, and outputs a set of aggregation keys $\mathbb{K}^{agg}$ of equal size, where element[11]

$$\mathbb{K}^{agg}[e] = \mathcal{H}_n(T_{agg}, \mathbb{S}^{base}, K_e^{s,base}) K_e^{s,base}$$

The aggregation private keys k_e^{agg} are used in group signatures.[12]

There is another operation `merge` which takes the aggregation keys from `premerge` and constructs the group signing key (e.g. spend key for Monero)

$$K^{grp} = \sum_e \mathbb{K}^{agg}[e]$$

We generalize these functions for (N-1)-of-N and M-of-N in Section 9.6.2, and further generalize them for nested multisig in Section 9.7.2.

[7] $\mathbb{S}^{base}$ needs to be ordered consistently so participants can be sure they are all hashing the same thing.

[8] Recalling Section 3.6, hash functions should be domain separated by prefixing them with tags, e.g. T_{agg} = "Multisig_Aggregation". We leave tags out for examples like the next section's Schnorr signatures.

[9] It is important to include $\mathbb{S}^{base}$ in the aggregation hashes to avoid sophisticated key cancellation attacks involving Wagner's generalized solution to the birthday problem [137]. [139] [90]

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

[11] Notation: $\mathbb{K}^{agg}[e]$ is the e^{th} element of the set.

[12] Robust key aggregation has not yet been implemented in Monero, but since participants can store and use private key k_e^{agg} (for naive key aggregation, $k_e^{agg} = k_e^{base}$), updating Monero to use robust key aggregation will only change the premerge process.

9.3 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 extend this to M-of-N (M $\leq$ N) multisig where N participants create the group address but only M people are needed to make signatures.

Let's take a step back from Monero. All signature schemes in this document lead from Maurer's general zero-knowledge proof of knowledge [87], so we can demonstrate the essential form of thresholded signatures using a simple Schnorr-like signature (recall Section 2.3.5) [109].

Signature

Say there are N people who each have a public key in the set $\mathbb{K}^{agg}$, where each person $e \in \{1, ..., N\}$ knows the private key k_e^{agg}. Their N-of-N group public key, which they will use to sign messages, is K^{grp}. Suppose they want to jointly sign a message $\mathfrak{m}$. 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 $\alpha_e \in_R \mathbb{Z}_l$,
   (b) computes $\alpha_e G$
   (c) commits to it with $C_e^\alpha = \mathcal{H}_n(T_{com}, \alpha_e G)$,
   (d) and sends C_e^α to the other participants securely.
2. Once all commitments C_e^α have been collected, each participant sends their $\alpha_e G$ to the other participants securely. They must verify that $C_e^\alpha \stackrel{?}{=} \mathcal{H}_n(T_{com}, \alpha_e G)$ for all other participants.
3. Each participant computes
$$\alpha G = \sum_e \alpha_e G$$
4. Each participant $e \in \{1, ..., N\}$ does the following:[13]
   (a) computes the challenge $c = \mathcal{H}_n(\mathfrak{m}, [\alpha G])$,
   (b) defines their response component $r_e = \alpha_e - c * k_e^{agg} \pmod l$,
   (c) and sends r_e to the other participants securely.
5. Each participant computes
$$r = \sum_e r_e$$
6. Any participant can publish the signature $\sigma(\mathfrak{m}) = (c, r)$.

[13] As in Section 2.3.4, it is important not to reuse α_e for different challenges c. This means to reset a multisignature process where responses have been sent out, it should start again from the beginning with new α_e values.

Verification

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

1. Compute the challenge $c' = \mathcal{H}_n(\mathfrak{m}, [rG + c * K^{grp}])$.
2. If $c = c'$ then the signature is legitimate except with negligible probability.

We included the superscript *grp* for clarity, but in reality the verifier has no way to tell K^{grp} is a merged key unless a participant tells him, or unless he knows the base or aggregation key components.

Why it works

Response r is the core of this signature. Participant e knows two secrets in r_e (α_e and k_e^{agg}), so his private key k_e^{agg} is information-theoretically secure from other participants (assuming he never reuses α_e). Moreover, verifiers use the group public key K^{grp}, so all key components are needed to build signatures.

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

Additional commit-and-reveal step

The reader may be wondering where Step 2 came from. Without commit-and-reveal [111], a malicious co-signer could learn all $\alpha_e G$ *before* the challenge is computed. This lets him control the challenge produced to some degree, by modifying his own $\alpha_e G$ prior to sending it out. He can use the response components collected from multiple controlled signatures to derive other signers' private keys k_e^{agg} in sub-exponential time [53], a serious security threat. This threat relies on Wagner's generalization [137] (see also [139] for a more intuitive explanation) of the birthday problem [141].[14]

[14] Commit-and-reveal is not used in the current implementation of Monero multisig, although it is being looked at for future releases. [102]

9.4 MLSTAG Ring Confidential signatures for Monero

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

Recalling Section 4.2.1, a one-time address assigning ownership of a transaction's t^{th} output to whoever has public address (K_t^v, K_t^s) goes like this

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

We can update our notation for outputs received by a group address $(K_t^{v,grp}, K_t^{s,grp})$:

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

Just as before, anyone with $k_t^{v,grp}$ and $K_t^{s,grp}$ can discover $K_t^{o,grp}$ is their address's owned output, and can decode the Diffie-Hellman term for output amount and reconstruct the corresponding commitment mask (Section 5.3).

This also means multisig subaddresses are possible (Section 4.3). Multisig transactions using funds received to a subaddress require some fairly straightforward modifications to the following algorithms, which we mention in footnotes.[15]

9.4.1 `RCTTypeBulletproof2` with N-of-N multisig

Most parts of a multisig transaction can be completed by whoever initiated it. Only the MLSTAG signatures require collaboration. An initiator should do these things to prepare for an `RCTTypeBulletproof2` transaction (recall Section 6.2):

1. Generate a transaction private key $r \in_R \mathbb{Z}_l$ (Section 4.2) and compute the corresponding public key rG (or multiple such keys if dealing with a subaddress recipient; Section 4.3).
2. Decide the inputs to be spent ($j \in \{1, ..., m\}$ owned outputs with one-time addresses $K_j^{o,grp}$ and amounts $a_1, ..., a_m$), and recipients to receive funds ($t \in \{0, ..., p-1\}$ new outputs with amounts $b_0, ..., b_{p-1}$ and one-time addresses K_t^o). This includes the miner's fee f and its commitment fH. Decide each input's set of decoy ring members.
3. Encode each output's amount $amount_t$ (Section 5.3), and compute the output commitments C_t^b.
4. Select, for each input $j \in \{1, ..., m-1\}$, pseudo output commitment mask components $x'_j \in_R \mathbb{Z}_l$, and compute the m^{th} mask as (Section 5.4)
$$x'_m = \sum_t y_t - \sum_{j=1}^{m-1} x'_j$$
Compute the pseudo output commitments $C_j'^a$.

[15] Multisig subaddresses are supported in Monero.

5. Produce the aggregate Bulletproof range proof for all outputs. Recall Section 5.5.

6. Prepare for MLSTAG signatures by generating, for the commitments to zero, seed components $\alpha_j^z \in_R \mathbb{Z}_l$, and computing $\alpha_j^z G$.[16]

He sends all this information to the other participants securely. Now the group of signers is ready to build input signatures with their private keys $k_e^{s,agg}$, and the commitments to zero $C_{\pi,j}^a - C_{\pi,j}'^a = z_j G$.

MLSTAG RingCT

First they construct the group key images for all inputs $j \in \{1, ..., m\}$ with one-time addresses $K_{\pi,j}^{o,grp}$.[17]

src/wallet/wallet2.cpp `sign_multisig_tx()`

1. For each input j each participant e does the following:
   (a) computes partial key image $\tilde{K}_{j,e}^o = k_e^{s,agg} \mathcal{H}_p(K_{\pi,j}^{o,grp})$,
   (b) and sends $\tilde{K}_{j,e}^o$ to the other participants securely.

2. Each participant can now compute, using u_j as the output index in the transaction where $K_{\pi,j}^{o,grp}$ was sent to the multisig address,[18]
$$\tilde{K}_j^{o,grp} = \mathcal{H}_n(k^{v,grp} rG, u_j) \mathcal{H}_p(K_{\pi,j}^{o,grp}) + \sum_e \tilde{K}_{j,e}^o$$

Then they build a MLSTAG signature for each input j.

1. Each participant e does the following:
   (a) generates seed components $\alpha_{j,e} \in_R \mathbb{Z}_l$ and computes $\alpha_{j,e} G$, and $\alpha_{j,e} \mathcal{H}_p(K_{\pi,j}^{o,grp})$,
   (b) generates, for $i \in \{1, ..., v+1\}$ except $i = \pi$, random components $r_{i,j,e}$ and $r_{i,j,e}^z$,
   (c) computes the commitment
$$C_{j,e}^\alpha = \mathcal{H}_n(T_{com}, \alpha_{j,e} G, \alpha_{j,e} \mathcal{H}_p(K_{\pi,j}^{o,grp}), r_{1,j,e}, ..., r_{v+1,j,e}, r_{1,j,e}^z, ..., r_{v+1,j,e}^z)$$
   (d) and sends $C_{j,e}^\alpha$ to the other participants securely.

src/wallet/wallet2.cpp `get_multisig_kLRki()`

[16] There is no need to commit-and-reveal these since the commitments to zero are known by all signers.

[17] If $K_{\pi,j}^{o,grp}$ is built from an i-indexed multisig subaddress, then (from Section 4.3) its private key is a composite:
$$k_{\pi,j}^{o,grp} = \mathcal{H}_n(k^{v,grp} r_{u_j} K^{s,grp,i}, u_j) + \sum_e k_e^{s,agg} + \mathcal{H}_n(k^{v,grp}, i)$$

[18] If the one-time address corresponds to an i-indexed multisig subaddress, add
$$\tilde{K}_j^{o,grp} = ... + \mathcal{H}_n(k^{v,grp}, i) \mathcal{H}_p(K_{\pi,j}^{o,grp})$$
src/cryptonote_basic/cryptonote_format_utils.cpp `generate_key_image_helper_precomp()`

2. Upon receiving all $C^{\alpha}_{j,e}$ from the other participants, send all $\alpha_{j,e}G$, $\alpha_{j,e}\mathcal{H}_p(K^{o,grp}_{\pi,j})$, and $r_{i,j,e}$ and $r^z_{i,j,e}$, and verify each participant's original commitment was valid.

3. Each participant can compute all
$$\alpha_j G = \sum_e \alpha_{j,e} G$$
$$\alpha_j \mathcal{H}_p(K^{o,grp}_{\pi,j}) = \sum_e \alpha_{j,e} \mathcal{H}_p(K^{o,grp}_{\pi,j})$$
$$r_{i,j} = \sum_e r_{i,j,e}$$
$$r^z_{i,j} = \sum_e r^z_{i,j,e}$$

4. Each participant can build the signature loop (see Section 3.5).

5. To finish closing the signature, each participant e does the following:
   (a) defines $r_{\pi,j,e} = \alpha_{j,e} - c_\pi k^{s,agg}_e \pmod l$,
   (b) and sends $r_{\pi,j,e}$ to the other participants securely.

6. Everyone can compute (recall $\alpha^z_{j,e}$ was created by the initiator)[19]
$$r_{\pi,j} = \sum_e r_{\pi,j,e} - c_\pi * \mathcal{H}_n(k^{v,grp} rG, u_j)$$
$$r^z_{\pi,j} = \alpha^z_{j,e} - c_\pi z_j \pmod l$$

src/ringct/rctSigs.cpp `signMultisig()`

The signature for input j is $\sigma_j(\mathfrak{m}) = (c_1, r_{1,j}, r^z_{1,j}, ..., r_{v+1,j}, r^z_{v+1,j})$ with $\tilde{K}^{o,grp}_j$.

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. The response $r_{\pi,j}$ is only useful for the $\mathfrak{m}$ and c_π it is defined for.

9.4.2 Simplified communication

It takes a lot of steps to build a multisignature Monero transaction. We can reorganize and simplify some of them so that signer interactions are encompassed by two parts with five total rounds.

1. *Key aggregation for a multisig public address*: Anyone with a set of public addresses can run `premerge` on them and then `merge` an N-of-N address, but no participant will know the group view key unless they learn all the components, so the group starts by sending k^v_e and $K^{s,base}_e$ to

[19] If the one-time address $K^{o,grp}_{\pi,j}$ corresponds to an i-indexed multisig subaddress, include
$$r_{\pi,j} = ... - c_\pi * \mathcal{H}_n(k^{v,grp}, i)$$
src/cryptonote_basic/cryptonote_format_utils.cpp `generate_key_image_helper_precomp()`

each other securely. Any participant can `premerge` and `merge` and publish $(K^{v,grp}, K^{s,grp})$, allowing the group to receive funds to the group address. M-of-N aggregation requires more steps, which we describe in Section 9.6.

src/wallet/wallet2.cpp `pack_multisignature_keys()`

2. *Transactions*:

   (a) Some participant or sub-coalition (the initiator) decides to write a transaction. They choose m inputs with one-time addresses $K_j^{o,grp}$ and amount commitments C_j^a, m sets of v additional one-time addresses and commitments to be used as ring decoys, pick p output recipients with public addresses (K_t^v, K_t^s) and amounts b_t to send them, decide a transaction fee f, pick a transaction private key r,[20] generate pseudo output commitment masks x'_j with $j \neq m$, construct the ECDH term $amount_t$ for each output, produce an aggregate range proof, and generate signature openers α_j^z for all inputs' commitments to zero and random scalars $r_{i,j}$ and $r_{i,j}^z$ with $i \neq \pi_j$.[21] They also prepare their contribution for the next communication round.

   src/wallet/wallet2.cpp `transfer_selected_rct()`

   The initiator sends all this information to the other participants securely.[22] The other participants can signal agreement by sending their part of the next communication round, or negotiate for changes.

   src/wallet/wallet2.cpp `save_multisig_tx()`

   (b) Each participant chooses their opening components for the MLSTAG signature(s), commits to them, calculates their partial key images, and sends those commitments and partial images to other participants securely.

   MLSTAG Signature(s): key image $\tilde{K}_{j,e}^o$, signature randomness $\alpha_{j,e}G$, and $\alpha_{j,e}\mathcal{H}_p(K_{\pi,j}^{o,grp})$. Partial key images don't need to be in committed data, as they can't be used to extract signers' private keys. They are also useful for viewing which owned outputs have been spent, so for the sake of modular design should be handled separately.

   (c) Upon receiving all signature commitments, each participant sends the committed information to the other participants securely.

   (d) Each participant closes their part of the MLSTAG signature(s), sending all $r_{\pi_j,j,e}$ to the other participants securely.[23]

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.

[20] Or transaction private keys r_t if sending to at least one subaddress.

[21] Note that we simplify the signing process by letting the initiator generate random scalars $r_{i,j}$ and $r_{i,j}^z$, instead of each co-signer generating components that eventually get summed together.

[22] He doesn't need to send the output amounts b_t directly, as they can be computed from $amount_t$. Monero takes the reasonable approach of creating a partial transaction filled with the information selected by the initiator, and sending that to other cosigners along with a list of related information like transaction private keys, destination addresses, the real inputs, etc.

[23] It is imperative that each signing attempt by a signer use a unique $\alpha_{j,e}$, to avoid leaking his private spend key to other signers (recall Section 2.3.4) [111]. Wallets should fundamentally enforce this by always deleting $\alpha_{j,e}$ whenever a response that uses it has been transmitted outside of the wallet.

src/wallet/wallet2.cpp `save_multisig_tx()`

9.5 Recalculating key images

If someone loses their records and wants to calculate their address's balance (received minus spent funds), they need to check the blockchain. View keys are only useful for reading received funds, so users need to calculate key images for all owned outputs to see if they have been spent, by comparing with key images stored in the blockchain. Since members of a group 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.[24] Given a received output with one-time address $K^{o,grp}$, the group can produce a simple 'linkable' Schnorr-like proof (basically single-key bLSTAG, recall Sections 3.1 and 3.4) to prove the key image $\tilde{K}^{o,grp}$ is legitimate without revealing their private spend key components or needing to trust each other.

Proof

1. Each participant e does the following:
   (a) computes $\tilde{K}_e^o = k_e^{s,agg}\mathcal{H}_p(K^{o,grp})$,
   (b) generates seed component $\alpha_e \in_R \mathbb{Z}_l$ and computes $\alpha_e G$ and $\alpha_e \mathcal{H}_p(K^{o,grp})$,
   (c) commits to the data with $C_e^\alpha = \mathcal{H}_n(T_{com}, \alpha_e G, \alpha_e \mathcal{H}_p(K^{o,grp}))$,
   (d) and sends C_e^α and $\tilde{K}_e^o$ to the other participants securely.
2. Upon receiving all C_e^α, each participant sends the committed information and verifies other participants' commitments were legitimate.
3. Each participant can compute:[25]
$$\tilde{K}^{o,grp} = \mathcal{H}_n(k^{v,grp} rG, u)\mathcal{H}_p(K^{o,grp}) + \sum_e \tilde{K}_e^o$$
$$\alpha G = \sum_e \alpha_e G$$
$$\alpha \mathcal{H}_p(K^{o,grp}) = \sum_e \alpha_e \mathcal{H}_p(K^{o,grp})$$
4. Each participant computes the challenge[26]
$$c = \mathcal{H}_n([\alpha G], [\alpha \mathcal{H}_p(K^{o,grp})])$$
5. Each participant does the following:

[24] Currently Monero appears to use a simple sum.

[25] If the one-time address corresponds to an i-indexed multisig subaddress, add
$$\tilde{K}^{o,grp} = ... + \mathcal{H}_n(k^{v,grp}, i)\mathcal{H}_p(K^{o,grp})$$
src/wallet/wallet2.cpp `export_multisig()`

[26] This proof should include domain separation and key prefixing, which we omit for brevity.

   (a) defines $r_e = \alpha_e - c * k_e^{s,agg} \pmod l$,

   (b) and sends r_e to the other participants securely.

6. Each participant can compute[27]

$$r^{resp} = \sum_e r_e - c * \mathcal{H}_n(k^{v,grp} rG, u)$$

The proof is (c, r^{resp}) with $\tilde{K}^{o,grp}$.

Verification

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

9.6 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. An escrow service provides 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 and day key to withdraw funds.

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

9.6.1 1-of-N key aggregation

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

1. One participant or sub-coalition selects a key and sends it to everyone else securely.

[27] If the one-time address $K^{o,grp}$ corresponds to an i-indexed multisig subaddress, include

$$r^{resp} = ... - c * \mathcal{H}_n(k^{v,grp}, i)$$

2. All participants compute private key components and send them securely, using the simple sum as the merged key.[28]

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 $(k^{v,grp,\text{1xN}}, k^{s,grp,\text{1xN}})$. Before this section all group keys were N-of-N, but now we use superscript 1xN to denote keys related to 1-of-N signing.

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

In an (N-1)-of-N group 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 base public keys K_e^{base} which they are all aware of.

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

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

Then he computes all $K_{e,w}^{sh,\text{(N-1)xN}} = k_{e,w}^{sh,\text{(N-1)xN}} G$ and sends them to the other participants securely. We now use superscript sh to denote keys shared by a sub-group of participants.

src/multisig/multisig.cpp `generate_multisig_N1_N()`

Each participant will have (N-1) shared 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 $\mathbb{S}^{\text{(N-1)xN}}$.

Generalizing `premerge` and `merge`

This is where we update the definition of `premerge` from Section 9.2.3. Its input will be the set $\mathbb{S}^{\text{MxN}}$, where M is the 'threshold' that the set's keys are prepared for. When $M = N$, $\mathbb{S}^{\text{NxN}} = \mathbb{S}^{base}$, and when $M < N$ it contains shared keys. The output is $\mathbb{K}^{agg,\text{MxN}}$.

The [N*(N-1)]/2 key components in $\mathbb{K}^{agg,\text{(N-1)xN}}$ can be sent into `merge`, outputting $K^{grp,\text{(N-1)xN}}$. Importantly, 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 N^{th} guy.

A 2-of-3 example

Say there are three people with public keys $\{K_1^{base}, K_2^{base}, K_3^{base}\}$, 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 each know the following:

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

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

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

Performing `premerge` and `merge` creates the group key:[29]

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

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 $\alpha_e \in_R \mathbb{Z}_l$,
   (b) computes $\alpha_e G$,
   (c) commits with $C_e^\alpha = \mathcal{H}_n(T_{com}, \alpha_e G)$,
   (d) and sends C_e^α to the other participants securely.
2. On receipt of all C_e^α, each participant sends out $\alpha_e G$ and verifies the other commitments were legitimate.
3. Each participant computes
$$\alpha G = \sum_e \alpha_e G$$
$$c = \mathcal{H}_n(\mathfrak{m}, [\alpha G])$$
4. Participant 1 does the following:
   (a) computes $r_1 = \alpha_1 - c * [k_{1,3}^{agg,2\text{x}3} + k_{1,2}^{agg,2\text{x}3}]$,
   (b) and sends r_1 to participant 2 securely.
5. Participant 2 does the following:
   (a) computes $r_2 = \alpha_2 - c * k_{2,3}^{agg,2\text{x}3}$,
   (b) and sends r_2 to participant 1 securely.
6. Each participant computes
$$r = \sum_e r_e$$

[29] Since the merged key is composed of shared secrets, an observer who just knows the original base public keys would not be able to aggregate them (Section 9.2.2) and identify members of the merged key.

7. Either participant can publish the signature $\sigma(\mathfrak{m}) = (c, r)$.

The only change with sub-N threshold signatures is how to 'close the loop' by defining $r_{\pi,e}$ (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 $\mathbb{S}^{base}$ of all participants' original keys, only the *first person* - ordered by index in $\mathbb{S}^{base}$ - with the copy of a shared secret uses it to calculate his $r_{\pi,e}$.[30,31]

In the previous example, participant 1 computes

$$r_1 = \alpha_1 - c * [k_{1,3}^{agg,2\text{x}3} + k_{1,2}^{agg,2\text{x}3}]$$

while participant 2 only computes

$$r_2 = \alpha_2 - c * k_{2,3}^{agg,2\text{x}3}$$

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

9.6.3 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^{base} and K_2^{base}, contains two private keys, $k_1^{base}k_2^{base}G$. It's a secret because only person 1 can compute $k_1^{base}K_2^{base}$, and only person 2 can compute $k_2^{base}K_1^{base}$.

What if there is a third person with K_3^{base}, there exist shared secrets $k_1^{base}k_2^{base}G$, $k_1^{base}k_3^{base}G$, and $k_2^{base}k_3^{base}G$, and the participants send those 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 $k_1^{base} * (k_2^{base}k_3^{base}G)$, person 2 computes $k_2^{base} * (k_1^{base}k_3^{base}G)$, and person 3 computes $k_3^{base} * (k_1^{base}k_2^{base}G)$. Now they all know $k_1^{base}k_2^{base}k_3^{base}G$, making a three-way shared secret (so long as no one publishes it). src/multisig/multisig.cpp `generate_multisig_derivations()`

The group could use $k^{sh,1\text{x}3} = \mathcal{H}_n(k_1^{base}k_2^{base}k_3^{base}G)$ as a shared private key, and publish

$$K^{grp,1\text{x}3} = \mathcal{H}_n(T_{agg}, \mathbb{S}^{1\text{x}3}, K^{sh,1\text{x}3})K^{sh,1\text{x}3}$$

as a 1-of-3 multisig address.

[30] In practice this means an initiator should determine which subset of M signers will sign a given message. If he discovers O signers are willing to sign, with (O ≥ M), he can orchestrate multiple concurrent signing attempts for each M-size subset within O to increase the chances of one succeeding. It appears Monero uses this approach. It also turns out (an esoteric point) that the *original* list of output destinations created by the initiator is randomly shuffled, and that random list is then used by all concurrent signing attempts, and all other co-signers (this is related to the obscure flag `shuffle_outs`). src/wallet/wallet2.cpp `transfer_selected_rct()`

[31] Currently Monero appears to use a round-robin signing method, where the initiator signs with all his private keys, passes the partially signed transaction to another signer who signs with all *his* private keys (that have not been used to sign with yet), who passes to yet another signer, and so on, until the final signer who can either publish the transaction or send it to other signers so they can publish it. src/wallet/wallet2.cpp `sign_multisig_tx()`

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 has a shared secret [109]. If (N-M) people are missing, all their shared secrets are owned by at least one of the M remaining people, who can collaborate to sign with the group's key.

M-of-N algorithm

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

src/wallet/wallet2.cpp `exchange_multisig_keys()`

We will use $\mathbb{S}_s$ to denote all the *unique* public keys at stage $s \in \{0, ..., (N-M)\}$. The final set $\mathbb{S}_{N-M}$ is ordered according to a sorting convention (such as smallest to largest numerically, i.e. lexicographically). This notation is a convenience, and $\mathbb{S}_s$ is the same as $\mathbb{S}^{\text{(N-s)xN}}$ from the previous sections.

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 $\mathbb{S}_{0,e}^K = \{K_e^{base}\}$.

The set $\mathbb{S}_e^k$ will contain each participant's aggregation private keys at the end.

1. Each participant e sends their original public key set $\mathbb{S}_{0,e}^K = \{K_e^{base}\}$ to the other participants securely.
2. Each participant builds $\mathbb{S}_0$ by collecting all $\mathbb{S}_{0,e}^K$ and removing duplicates.
3. For stage $s \in \{1, ..., (N-M)\}$ (skip if M = N)
   (a) Each participant e does the following:
      i. For each element g_{s-1} of $\mathbb{S}_{s-1} \notin \mathbb{S}_{s-1,e}^K$, compute a new shared secret
      $$k_e^{base} * \mathbb{S}_{s-1}[g_{s-1}]$$
      ii. Put all new shared secrets in $\mathbb{S}_{s,e}^K$.
      iii. If $s = (N-M)$, compute the shared private key for each element x in $\mathbb{S}_{N-M,e}^K$
      $$\mathbb{S}_e^k[x] = \mathcal{H}_n(\mathbb{S}_{N-M,e}^K[x])$$
      and overwrite the public key by setting $\mathbb{S}_{N-M,e}^K[x] = \mathbb{S}_e^k[x] * G$.
      iv. Send the other participants $\mathbb{S}_{s,e}^K$.
   (b) Each participant builds $\mathbb{S}_s$ by collecting all $\mathbb{S}_{s,e}^K$ and removing duplicates.[32]
4. Each participant sorts $\mathbb{S}_{N-M}$ according to the convention.
5. The `premerge` function takes $\mathbb{S}_{(N-M)}$ as input, and each aggregation key is, for $g \in \{1, ..., (\text{size of } \mathbb{S}_{N-M})\}$,
$$\mathbb{K}^{agg,\text{MxN}}[g] = \mathcal{H}_n(T_{agg}, \mathbb{S}_{(N-M)}, \mathbb{S}_{(N-M)}[g]) * \mathbb{S}_{(N-M)}[g]$$

[32] 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 9.6.2.

6. The `merge` function takes $\mathbb{K}^{agg,\text{MxN}}$ as input, and the group key is

$$K^{grp,\text{MxN}} = \sum_{g=1}^{\text{size of } \mathbb{S}_{N-M}} \mathbb{K}^{agg,\text{MxN}}[g]$$

7. Each participant e overwrites each element x in $\mathbb{S}_e^k$ with their aggregation private key

$$\mathbb{S}_e^k[x] = \mathcal{H}_n(T_{agg}, \mathbb{S}_{(N-M)}, \mathbb{S}_e^k[x]G) * \mathbb{S}_e^k[x]$$

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

9.7 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.

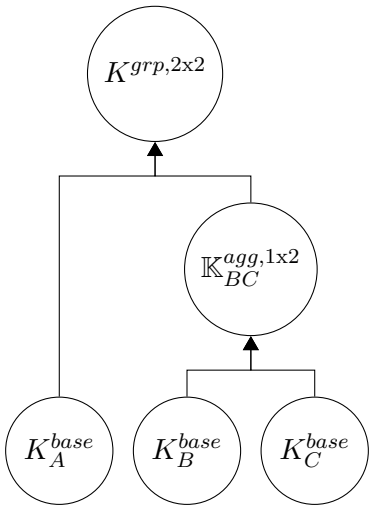
Now imagine Alice wants to sign all transactions from that address, but doesn't want Bob and Carol to sign without her. In other words, (Alice + Bob) or (Alice + Carol) are acceptable, but not (Bob + Carol).

We can achieve that scenario with two layers of key aggregation. First a 1-of-2 multisig aggregation $\mathbb{K}_{BC}^{agg,\text{1x2}}$ between Bob and Carol, then a 2-of-2 group key $K^{grp,\text{2x2}}$ between Alice and $\mathbb{K}_{BC}^{agg,\text{1x2}}$. Basically, a (2-of-([1-of-1] and [1-of-2])) multisig address.

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

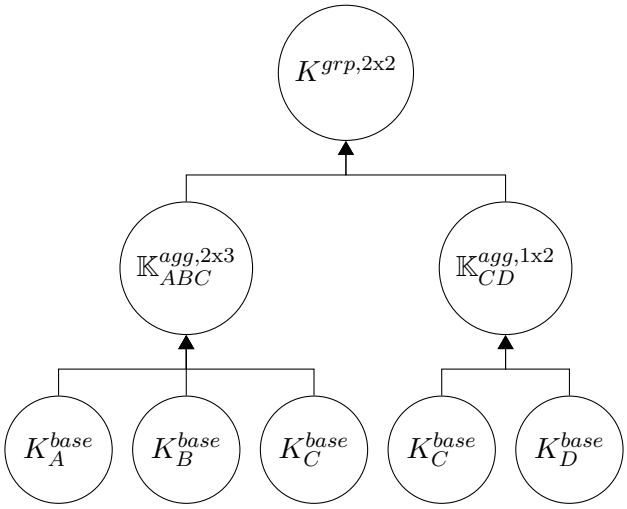
9.7.1 Family trees

We can diagram the (2-of-([1-of-1] and [1-of-2])) multisig address like this:



The keys $K_A^{base}, K_B^{base}, K_C^{base}$ are considered *root ancestors*, while $\mathbb{K}_{BC}^{agg,1\text{x}2}$ is the *child* of *parents* K_B^{base} and K_C^{base}. 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.

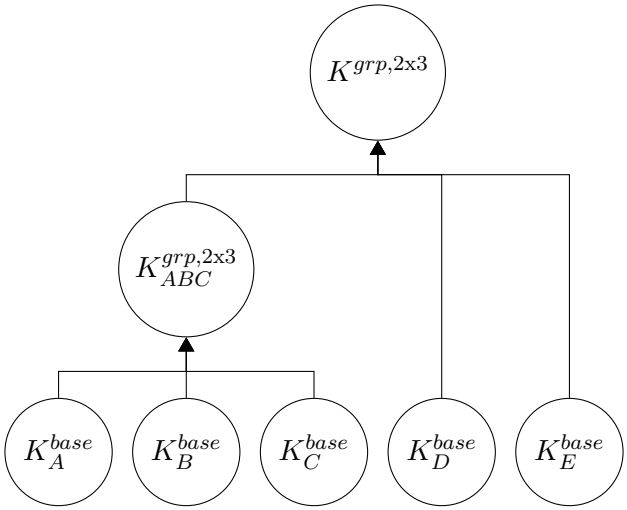
For example, in this 2-of-3 and 1-of-2 joined in a 2-of-2, Carol's key K_C^{base} is used twice and displayed twice:



Separate sets $\mathbb{S}$ are defined for each multisig sub-coalition. There are three premerge sets in the previous example: $\mathbb{S}_{ABC}^{2\text{x}3} = \{K_{AB}^{sh,2\text{x}3}, K_{BC}^{sh,2\text{x}3}, K_{AC}^{sh,2\text{x}3}\}$, $\mathbb{S}_{CD}^{1\text{x}2} = \{K_{CD}^{sh,1\text{x}2}\}$, and $\mathbb{S}_{final}^{2\text{x}3} = \{\mathbb{K}_{ABC}^{agg,2\text{x}3}, \mathbb{K}_{CD}^{agg,1\text{x}2}\}$.

9.7.2 Nesting multisig keys

Suppose we have the following key family



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

$$k_{ABC,D} = \mathcal{H}_n(k_{ABC}^{grp,2\text{x}3} K_D^{base})$$

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

$$k_{ABC}^{grp,2\text{x}3} K_D^{base} = \sum k_{ABC}^{agg,2\text{x}3} K_D^{base}$$

The problem is everyone from ABC can compute $k_{ABC,D}^{sh,2\text{x}3} = \mathcal{H}_n(k_{ABC}^{grp,2\text{x}3} K_D^{base})$! 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 `premerge` for all keys output by low-tier multisigs.

Solution for nesting

To use $\mathbb{K}^{agg,\text{MxN}}$ in a new multisig, we pass it around just like a normal key, with one change. Operations involving $\mathbb{K}^{agg,\text{MxN}}$ use each of its member keys, instead of the merged group key. For example, the public 'key' of a shared secret between $\mathbb{K}_x^{agg,2\text{x}3}$ and K_A^{base} would look like

$$\mathbb{K}_{x,A}^{sh,1\text{x}2} = \{[\mathcal{H}_n(k_A^{base} \mathbb{K}_x^{agg,2\text{x}3}[1]) * G], [\mathcal{H}_n(k_A^{base} \mathbb{K}_x^{agg,2\text{x}3}[2]) * G], ...\}$$

This way all members of $\mathbb{K}_x^{agg,2\text{x}3}$ only know shared secrets corresponding to their private keys from the lower-tier 2-of-3 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 (e.g. $\mathbb{S}$ sets) are ordered by the first element in each keyset, according to some convention.

We let the key sets propagate through the family structure, with each nested multisig group sending their `premerge` aggregation set as the 'base key' for the next level,[33] until the last child's aggregation set appears, at which point `merge` is finally used.

Users should store their base private keys, the aggregation private keys for all levels of a multisig family structure, and the public keys for all levels. Doing so facilitates creating new structures, `merging` nested multisigs, and collaborating with other signers to rebuild a structure if there is a problem.

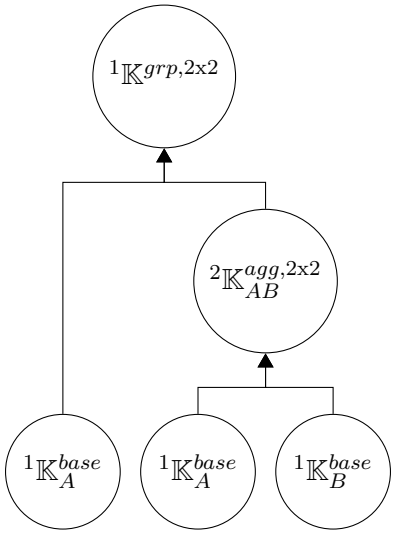
[33] Note that `premerge` needs to be done to the outputs of *all* nested multisigs, even when an N'-of-N' multisig is nested into an N-of-N, because the set $\mathbb{S}$ will change.

9.7.3 Implications for Monero

Each sub-coalition contributing to the final key needs to contribute components to Monero transactions (such as the opening values αG), 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 group value x for a signature. Root ancestors contribute: $x_{A,1}$, $x_{A,2}$, x_B. The total is $x = x_{A,1} + x_{A,2} + x_B$.

There are currently no implementations of nested multisig in Monero.

CHAPTER 10

Monero in an Escrowed Marketplace

Most of the time purchasing from an online retailer goes smoothly. A buyer sends money to a vendor, and the expected product arrives at their doorstep. If the product has a defect, or in some cases if the buyer changes their mind, they can return it and get a refund.

It's difficult to trust a person or organization you have never met, and many shoppers can feel safe in their purchases knowing their credit card company will reverse a payment on request [30].

Cryptocurrency transactions are not reversible, and there is limited legal recourse a buyer or seller can take when something goes wrong, especially for Monero which is not open to easy chain analysis [48]. The cornerstone of robust online shopping with cryptocurrencies is 2-of-3 multisignature-based escrowed exchanges, which enable third parties to mediate disputes. By trusting those third parties, even completely anonymous vendors and shoppers can interact without formality.

Since Monero multisig interactions can be rather complex (recall Section 9.4.2), we dedicate this chapter to describing a maximally efficient escrowed shopping environment.[1,2]

10.1 Essential features

There are several basic requirements and features for streamlined interactions between buyers and sellers online. We take these points from René's OpenBazaar investigation [117], as they are quite

[1] René "rbrunner7" Brunner, a Monero contributor who created the MMS [116, 115], investigated integrating Monero multisig into the decentralized cryptocurrency-based digital marketplace OpenBazaar `https://openbazaar.org/`. The concepts laid out here are inspired by the hurdles René encountered [117] (his 'earlier analysis').

[2] Our initial impression is Monero's current implementation of multisig already has a similar transaction creation flow to what we need for an escrowed shopping environment, which is good news for potential implementation efforts. Readers should note that Monero multisig requires some security updates before it can see heavy use [102].

sensible and extensible.

- *Offline selling*: A buyer should be able to access a vendor's online shop and place an order while the vendor is offline. Obviously, the vendor must actually go online to receive the order and fulfill it.[3]
- *Purchase order-based payments*: Vendor addresses for receipt of funds are unique for each purchase order, in order to match orders and payments.
- *High-trust purchasing*: A buyer may, if they trust a vendor, pay for a product even before it has been fulfilled or delivered.
  - *Online direct payment*: After confirming the vendor is online, and that their product listing is available, the purchaser sends money in a single transaction to the vendor via a vendor-provided address, who then signals the order is being fulfilled.
  - *Offline payment*: If a vendor is offline, the buyer creates and funds a 1-of-2 multisig address with enough money to cover their intended purchase. When the vendor comes online, they may take money out of the multisig address (sending change back to the buyer), and fulfill the order. If the vendor never comes back online (or e.g. after some reasonable waiting period), or if the buyer changes their mind before he comes back, the buyer may empty the 1-of-2 address back into her personal wallet.
- *Moderated purchasing*: A 2-of-3 multisig address is constructed between the buyer, vendor, and a moderator agreed on by both buyer and vendor. The buyer funds this address before vendor fulfillment, and then once the product is delivered two of the parties cooperate to release the funds. If the vendor and buyer can't reach an agreement, they may enlist the moderator's judgement.

We will not cover high-trust purchasing, as, without any complex communication required, the features are fairly trivial.[4]

10.1.1 Purchasing workflow

All purchases should fit within the same set of steps, assuming all parties do their due diligence. A number of steps are related to what a moderator should expect when stepping in, e.g. "did you request a refund, before getting me involved?".

1. A buyer accesses the vendor's online shop, identifies a purchase to make, selects 'I want to purchase it', makes funding available for that purchase, then submits the purchase order to the vendor.

[3] Fulfilling a purchase order means sending the product out to be delivered to the purchaser.

[4] 1-of-2 multisig can take advantage of some concepts useful for 2-of-3 multisig, in particular constructing the address in the first place.

2. The vendor receives the purchase order, verifies the product is in stock and the available funding is enough, and either returns money to the buyer or fulfills the order by shipping out the product and notifying the buyer with a receipt.
    - For a 2-of-3 multisig, the buyer can optionally authorize payment when receiving notification of fulfillment.

3. The buyer either receives the product as expected, or doesn't receive the product on time or receives a defective product.
    - *Good product*: The buyer either gives feedback to the vendor, or does nothing.
        - *Buyer sends feedback*: The buyer can leave either positive, or negative feedback.
            - *Positive feedback*: If this is a 2-of-3 multisig that hasn't been paid yet, then this is the step where the buyer confirms payment to the vendor. Otherwise it's just a positive review. [END OF WORKFLOW]
            - *Negative feedback*: If this is a 2-of-3 multisig, then this step leads into the 'bad product' workflow. Otherwise it's just a negative review.
        - *Buyer does nothing*: Either the vendor has already been paid, or it's a 2-of-3 multisig and he needs to cooperate with someone to receive funds.
            - *Vendor has been paid*: [END OF WORKFLOW]
            - *Vendor has not been paid*: The vendor pursues payment.
            (a) The vendor contacts the buyer requesting payment (or sends out a reminder).
            (b) The buyer either responds, or does not respond.
                - *Buyer responds*: The buyer's response can be to make a payment, or pursue a refund.
                > *Buyer makes payment*: [END OF WORKFLOW]
                > *Buyer pursues refund*: Go to the 'bad product' workflow.
                - *Buyer does not respond*: The vendor enlists the moderator's help to release funds. [END OF WORKFLOW]
    - *No product or bad product*: The buyer pursues a refund.
        (a) The buyer contacts the vendor requesting a refund, perhaps with an explanation.
        (b) The vendor complies with the refund request, or contests it (or ignores it).
            - *Vendor complies*: Money is refunded to the buyer. [END OF WORKFLOW]
            - *Vendor contests*: Either it's not a 2-of-3 multisig, or it is.
                - *It's not a 2-of-3*: [END OF WORKFLOW]
                - *It's a 2-of-3*: Either the buyer gives up on the refund request (literally, or by failing to respond in a timely manner), or the buyer is adamant.
                    - *Buyer gives up*: Either the buyer authorized the vendor's payment, or not.
                    > *Buyer authorized payment*: [END OF WORKFLOW]
                    > *Buyer didn't authorize*: The vendor contacts the moderator, who authorizes the payment. [END OF WORKFLOW]
                    - *Buyer is adamant*: The vendor or buyer contacts the moderator, who cooperates with the participants to pass judgement. [END OF WORKFLOW]

10.2 Seamless Monero multisig

We can take advantage of the natural purchase order workflow to squeeze in nearly all parts of a Monero 2-of-3 multisignature interaction without the participants even noticing. There is an extra small step for the vendor at the end, where he must sign and submit the final transaction to receive payment, analogous to 'emptying the cash register'.[5]

10.2.1 Basics of a multisig interaction

All 2-of-3 multisig interactions contain the same set of communication rounds, involving address setup and transaction building. In order to comply with the normal workflow we use a reorganized transaction construction process compared to our multisig chapter (recall Sections 9.4.2 and 9.6.2).[6]

1. *Address setup*
   (a) All users must first learn the other participants' base keys, which they will use to construct shared secrets. They transmit the public keys of those shared secrets (e.g. $K^{sh} = \mathcal{H}_n(k_A^{base} * k_B^{base} G)G$) to the other users.
   (b) After learning all shared secret public keys, each user may `premerge` and then `merge` them into the address's public spend key. The aggregation private spend keys will be used for signing transactions. A hash of the primary signers' (the buyer and vendor) shared secret private key (e.g. $k^{sh} = \mathcal{H}_n(k_A^{base} * k_B^{base} G)$) will be used as the view key (e.g. $k^v = \mathcal{H}_n(k^{sh})$). We go into more detail on these keys later (Section 10.2.2).
2. *Transaction building*: Assume the address owns at least one output already, and the key images are unknown. There are two signers, the initiator and the cosigner.
   (a) *Initiate a transaction*: The initiator decides to start a transaction. She generates opening values (e.g. αG) for all owned outputs (she isn't sure which ones will get used yet), and commits to them. She also creates partial key images for those owned outputs, and signs them with a proof of legitimacy (recall Section 9.5). She sends that information, along with her own personal address for receipt of funds (e.g. for change if appropriate, etc.), to the cosigner.
   (b) *Make a partial transaction*: The cosigner verifies that all information in the initiated transaction is valid. He decides the output recipients and amounts (they could be partially based on the initiator's recommendation), inputs to be used along with their respective ring member decoys, and the transaction fee. He generates a transaction private key (or keys if there are subaddresses involved), creates one-time addresses,

[5] Extending this beyond (N-1)-of-N is likely infeasible without more steps, due to the additional rounds necessary for setting up a lower threshold multisig address.
[6] This procedure is actually quite similar to how Monero currently organizes multisig transactions.

output commitments, encoded amounts, pseudo output commitment masks, and opening values for the commitments to zero. To prove amounts are in range, he builds the Bulletproof range proof for all outputs. He also generates opening values for his signature (but does not commit to them), random scalars for the MLSAG signatures, partial key images for owned outputs, and proofs of legitimacy for those partial images. All of this is sent to the initiator.

(c) *Initiator's partial signature*: The initiator verifies the partial transaction's information is valid, and conforms with her expectations (e.g. amounts and recipients are as they should be). She completes the MLSAG signatures and signs them with her private keys, then sends the partially signed transaction to the cosigner along with her revealed opening values.

(d) *Finish the transaction*: The cosigner finishes signing the transaction, and submits it to the network.

Single-commitment signing

Unlike what is recommended in our multisig chapter, only one commitment is provided per partial transaction (by the transaction initiator), and it is revealed after the cosigner sends out their opening value explicitly. The purpose of committing to opening values (e.g. αG) is to prevent a malicious cosigner from using their own opening value to affect the challenge that gets produced, which could allow him to discover other cosigners' aggregation keys (recall Section 9.3). If even one partial opening value is unavailable when the malicious actor generates his own, then it is impossible (or at least negligibly probable) for him to control the challenge.[7,8,9]

Simplifying in this way removes one communication round, which has important consequences for the buyer-vendor interaction experience.

[7] This is also why the initiator only reveals her opening values after all transaction information has been determined, so neither signer can alter the MLSAG message and influence the challenge.

[8] Single-commitment signing could be generalized as (M-1)-commitment signing, where only the partial transaction author does not commit-and-reveal and other co-signers only reveal after a transaction is fully determined. For example, suppose there is a 3-of-3 address with cosigners (A,B,C), who will attempt single-commitment signing. Signers B and C are in a malicious coalition against A, while C is the initiator and B is the partial transaction author. C initiates with a commitment, then A provides his opening value (without commitment). When B constructs the partial transaction, he can conspire with C to control the signature challenge in order to expose A's private key. Please also note that (M-1)-commitment signing is an original concept first presented here, and is not backed by any advanced research material or code implementation. It may end up being completely erroneous.

[9] One way of thinking about this is to consider the meaning and purpose of a 'commitment' (recall Section 5.1). Once Alice commits to value A she is stuck with it, and can't take advantage of new information from event B (caused by Bob) that happens later. Moreover, if A hasn't been revealed then B can't be influenced by it. Alice and Bob can be sure that A and B are independent. It is our contention that single-commitment signing as described meets that standard, and is equivalent to full-commitment signing. If the commitment c is a one-way function of opening values $\alpha_A G$ and $\alpha_B G$ (e.g. $c = \mathcal{H}_n(\alpha_A G, \alpha_B G)$), then if $\alpha_A G$ is committed to initially, $\alpha_B G$ is revealed after $C(\alpha_A G)$ appears, and $\alpha_A G$ is revealed after $\alpha_B G$ appears, then $\alpha_B G$ and $\alpha_A G$ are independent, and c will be random from both Alice's and Bob's perspectives (unless they collaborate, and except with negligible probability).

10.2.2 Escrowed user experience

Here is our detailed walk-through of buyer-vendor-moderator interactions involving a 2-of-3 multisig-based online purchase using Monero.

1. *Buyer makes a purchase*

   (a) *Buyer's new shopping session*: A shopper enters an online marketplace, and her client generates a new subaddress to be used if she starts a new purchase order.[10] In that marketplace she finds vendors, and each vendor is offering a selection of products and prices. Invisible to the shopper herself, but visible to her client (i.e. the software she is using to buy things), each product has a base key for use in multisig-based purchases. Alongside that base key is a list of preselected moderators, and each such moderator has a base key and precomputed vendor-moderator shared secret public key.[11]

   (b) *Buyer adds a product to cart*: Our shopper decides she wants something, selects the payment option (i.e. direct payment, 1-of-2 multisig, or 2-of-3 multisig), and if she selects 2-of-3 multisig she is presented with a list of available moderators that she can choose from. When she adds the product to her cart, her client, invisible to her (and assuming she selected 2-of-3 multisig), uses the product's base key, the moderator's base key, and the vendor-moderator shared secret public key, to, in combination with her own shopping-session subaddress's spend key (as her base key), construct a 2-of-3 buyer-vendor-moderator multisig address.[12]

      The view key is a hash of the buyer-vendor shared secret private key (not the aggregation private key, i.e. before `premerge`), and the encryption key for communications between the buyer and vendor is a hash of the view key.[13]

   (c) *Buyer moves to checkout*: The shopper views her cart with all its products, and decides to proceed to checkout. This is where she makes funding available before finalizing the purchase order. Her client constructs a transaction (but does not sign it yet) that will either pay directly to the vendor, or fund a multisig address (with a little extra for future fees). If funding a 2-of-3 multisig address, the client also initiates two transactions sending money out of that address. One can be used to pay the vendor, and the other to refund the buyer. Their inputs' partial key images are based on the funding transaction that has not been signed yet.

[10] Using a new subaddress for each purchase order, or even a new subaddress for each separate vendor or vendor's product, makes it more difficult for vendors to track the behavior of their customers. It also helps make sure each purchase order is unique, in the case of someone buying the same thing twice.

[11] It would be straightforward for vendors to also include, invisible to shoppers, commitments to opening values for transactions. However, to handle multiple purchase orders for the same product, he would have to provide many commitments up front for each potential buyer. We can only imagine how messy that could become. This is part of the utility brought by our single-commitment signing simplification.

[12] Exactly how a marketplace should be implemented is open to interpretation, since for example selecting the payment type for a product could be presented to the user at checkout instead of in the 'add to cart' interface.

[13] This same process would take place for 1-of-2 multisig, leaving out the moderator.

In reality, she only needs committed opening values for the two transactions, and then separately one copy of the partial key images (with proof of legitimacy) and one copy of her shopping-session subaddress. That subaddress has a dual purpose; it is the buyer's address for refunds or change outputs, and its spend key is the buyer's multisig base key.[14]

(d) *Buyer authorizes payment*: After looking over all the purchase order details, the shopper authorizes it.[15] Her client finishes signing the funding transaction, and submits it to the network.[16] It sends the purchase order, along with the funding transaction's hash and initiated multisig transactions, and the buyer-moderator shared secret public key, to the vendor.[17]

2. *Vendor fulfills a purchase order*

(a) *Vendor appraises purchase order*: The vendor examines our shopper's purchase order, then approves it for shipping. If he was paid directly then there is nothing else to consider, and if he was paid via 1-of-2 multisig then he can make a transaction paying himself out of that address. For 2-of-3 multisig his client generates a subaddress for receipt of money in the purchase order, and makes two partial transactions out of the buyer's initialized transactions. The payment transaction sends the product price to the vendor, and the rest to the buyer as change, while the refund transaction just empties the multisig address back to the buyer.[18] Note that he reconstructs the multisig address out of the buyer-vendor-moderator base keys in combination with the buyer-moderator shared secret public key.

(b) *Vendor ships the product*: The vendor ships the product, and sends a fulfillment notification to the buyer. That notification includes a receipt for the purchase, as well as a request for the user to complete payment (everything from here on out refers to 2-of-3 multisig). Hidden to the user is the vendor's purchase order subaddress, which can be used in case of dispute resolution, and both partial transactions.

3. *Buyer completes payment or requests a refund*: A buyer can do this as soon as they receive fulfillment notification, or wait until the product has been delivered.

(a) *Buyer submits partially signed transaction*: The buyer decides whether to pay for her purchase, or request a refund. Her client creates a partial signature on the appropriate partial transaction, and sends it to the vendor. Any refund would presumably include an explanation justifying it.

[14] It is important to initiate separate transactions, since committed opening values can only be used once.

[15] If the buyer cancels the purchase order, her funding transaction and partial multisig transactions get deleted.

[16] If her cart contained multiple vendors' products, then her client can create multiple purchase orders and handle them separately. The vendors can all be paid by the same funding transaction.

[17] The buyer's client should keep track of payment order details like total price, to later verify the content of multisig transactions before signing them.

[18] The partial transactions could plausibly share a lot of values since they involve the same inputs, and only one of them should ultimately get signed. For the sake of modularity and robust design we think it's best to handle them separately.

(b) *Vendor completes transaction*: The vendor receives a partially signed transaction, finishes signing it, and submits it to the network. If necessary he sends a refund notification, with proof, to the buyer.

4. *Moderated dispute*: At any point after the buyer submits a purchase order and before the multisig address is emptied of funds, either the vendor or buyer can decide to get the moderator involved. Party_A is whoever raised the dispute, while Party_B is the defendant.[19]

   (a) *Party_A contacts moderator*: Party_A initiates two transactions, for payment or refund, this time geared toward Party_A-moderator signing, and send them to the moderator along with necessary information for building the multisig address (the base keys, the Party_A-Party_B shared secret public key, and the private view key) and reading the multisig wallet's balance (the partial key images and their proofs).

   (b) *Moderator processes dispute*

      i. *Moderator acknowledges dispute*: The moderator acknowledges they are looking into the dispute, at the same time creating partial transactions out of the initiated transactions and sending them to Party_A. They make sure to notify Party_B about the dispute, and also initiate two more transactions with Party_B to expedite failure of Party_A to comply with the final verdict.

      ii. *Moderator pursues a verdict*: The moderator looks at the available evidence, and can interact with the parties to gather more information. They may try to mediate the argument, in hopes of both parties resolving it without the need to pass a verdict.

      iii. *Dispute reaches an end*: Either the buyer and vendor resolve it on their own in the end, or the moderator passes a verdict which they communicate to both parties.

      - Note: If the defendant party should receive funds per the verdict, but hasn't provided an address for whatever reason, the moderator can try to contact them and cooperate with them to receive those funds. Since it doesn't involve the disputer, that contact can be made (or pursued further) after the dispute has been resolved.

   (c) *Party_A or B accepts the verdict*: If no Party_A-Party_B transactions were finalized, it implies the dispute was resolved by moderator's decision.

      i. *Party_A accepts*

         A. Party_A completes their partial signature on the verdict transaction, and sends it to the moderator.

         B. The moderator completes the signature, and submits the transaction.

      ii. *Party_B accepts*

         A. Party_B creates a partial transaction for the moderator's initialized verdict transaction, and sends it to the moderator. This step can be performed before

[19] Our dispute resolution design assumes actors are in good faith. People who are uncooperative, and for example don't initiate or sign transactions that don't favor themselves, will no doubt make the process much more tedious for everyone.

the verdict is finalized, in which case Party_B would make partial transactions for both potential verdicts.

B. The moderator partially signs that partial transaction, and sends it to Party_B.

C. Party_B finishes signing the transaction, and submits it to the network. He sends the transaction hash to the moderator.

(d) *Moderator closes out the dispute*: The moderator summarizes the dispute and its resolution, and sends their report to the buyer and vendor.

There are four key design optimizations installed.

Preselected moderators

By choosing the moderators ahead of time, vendors can create a shared secret with each of them for each of their products and publish its public key with the product information.[20] This way buyers can construct the complete merged multisig address in one step, as soon as they decide to purchase something, harmonizing with the requirement for 'offline selling'. Preselecting multiple moderators allows buyers to choose the one they trust more.

Buyers may also, if they don't trust a vendor's accepted moderators, cooperate with an online moderator of their choosing to make a multisig address with the vendor's product base key. After receiving a purchase order, the vendor may accept that new moderator or decline the sale.[21]

We expect the mutual desire of buyers and vendors for good moderators to, over time, create a hierarchy of moderators organized by the quality and fairness of the service they provide. Lower quality moderators or those with less of a reputation would likely earn less money or service fewer or less significant transactions.[22]

Subaddresses and product IDs

Vendors create a new base key for each product line/ID, and those keys are used to construct 2-of-3 multisig addresses.[23] When vendors fulfill a purchase order they create a unique subaddress for receipt of funds, which can be used to match purchase orders with payments received.

[20] Importantly, these multisig addresses are still resistant to key aggregation tests since the buyer's shared secrets are unknown to observers.

[21] For expediency, an escrow service could be 'always online', and instead of using preselected moderators all 2-of-3 multisig addresses are actively constructed with that service when a purchase order is made. Another possibility is using nested multisig (Section 9.7), where the preselected moderator is actually a 1-of-N multisig group. That way whenever a dispute arises any moderator from that group who happens to be available can step in. It would likely require substantial development effort to implement.

[22] It is not clear to us the best, or most likely, funding method for moderators. Perhaps they would get paid a flat or percentage fee of each moderated transaction or transaction where they are added as a moderator (and then if the fee wasn't provided in the original partial transactions, refuse to cooperate in a dispute), or users and/or vendors and/or marketplace platforms would contract with them.

[23] This base key is also used for 1-of-2 multisig purchasing. We feel it is important not to expose the private spend key in the communication channel, so using a buyer-vendor shared secret makes a lot of sense.

The requirement for 'purchase order-based payments' is met efficiently here, especially since funds directed to different subaddresses are trivially accessible from the same wallet (recall Section 4.3).

Anticipatory partial transactions

Multisig transactions take multiple rounds, so we begin making them as soon as possible. For user convenience, partial transactions that are only rarely used (e.g. refund transactions) get made early, so they are immediately available for signing if the need arises.

Conditional moderator access

For the sake of both efficiency and privacy, moderators only need access to the details of a trade when settling disputes. To accomplish this, we make the multisig private view key a hash of the buyer-vendor shared secret private key $k^{v,grp}_{purchase-order} = \mathcal{H}_n(T_{mv}, k^{sh,2x3}_{AB})$ where T_{mv} is the marketplace view key domain separator and A and B correspond with vendor and buyer, and $k^{sh,2x3}_{AB} = \mathcal{H}_n(k^{base}_A * k^{base}_B G)$. We extend what it can 'view' to the buyer-vendor communication transcript. In other words, the communication encryption key is $k^{ce}_{purchase-order} = \mathcal{H}_n(T_{me}, k^{v,grp}_{purchase-order})$ (T_{me} is the marketplace encryption key domain separator).[24,25]

Moderators gain access to the buyer-vendor communications, and the ability to authorize payments, only when one of the original parties releases the view key to them.[26]

Moreover, vendors can verify the marketplace host (who may also be the only moderator available, depending on how this concept is implemented) is not MITM ('man in the middle') of their conversations with customers (i.e. pretending to be the buyer or vendor) by checking that the base keys they publish for each product match what gets displayed. Since the buyer's base key, which gets used to create the multisig address, is also part of the encryption key, a malicious host would have a hard time orchestrating a MITM attack.

[24] Separating the view key and encryption key allows handing out just view rights to the communication log without view rights to the multisig address's transaction history.

[25] This method is also used for 1-of-2 multisig addresses.

[26] It is important for moderators to verify the communication log they receive hasn't been tampered with. One way might be for each cosigner to include a signed hash of the current message log whenever they send out a new message. Moderators can look at the back-and-forth, and series of hashed logs, to identify discrepancies. It would also help cosigners identify messages that failed to transmit, and alternatively create evidence that particular signers did in fact receive certain messages.

CHAPTER 11

Joint Monero Transactions (TxTangle)

There are a number of unavoidable transaction graph heuristics created by the nature of different entities and scenarios. In particular, the behavior of miners, pools (Section 5.1 of [93]), escrowed marketplaces, and exchanges have clear patterns open to analysis even within Monero's ring signature-based protocol.

We describe here TxTangle, analogous to Bitcoin's CoinJoin [2], one method to confuse those heuristics.[1] In essence, several transactions are squashed into one transaction, making the behavior patterns of each participant blend together.

To accomplish that obfuscation, it must be unreasonably difficult for observers to use the information contained in a joint transaction to group inputs and outputs, and associate them with individual participants, nor know how many participants there actually were.[2] Moreover, even the participants themselves should not be aware of the number of participants, or be able to group the inputs and outputs of other participants unless they control all but one participant's grouping.[3] Finally, it should be possible to construct joint transactions without relying on a central authority [58]. Fortunately, all these requirements can be achieved in Monero.

[1] This chapter constitutes the proposal for a joint transaction protocol. No such protocol has been implemented as of this writing. A previous proposal, named MoJoin and created by pseudonymous co-author Monero Research Lab (MRL) researcher Sarang Noether, required a trusted dealer to function. Such a dealer seems to conflict with the Monero project's basic commitment to privacy and fungibility, and hence MoJoin was not pursued further.

[2] Since in Bitcoin amounts are clearly visible, it is often possible to group CoinJoin inputs and outputs based on amount sums. [34]

[3] Maliciously polluting joint transactions is a potential attack on this method, first identified for CoinJoin. [91]

11.1 Building joint transactions

In a normal transaction, inputs and outputs are tied together using the proof that amounts balance. From Section 6.2.1, the sum of pseudo output commitments equals the sum of output commitments (plus the fee commitment).

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

A trivial joint transaction could take all the content of multiple transactions, and stick them in one. MLSAG messages would sign all the sub-transactions' data, and amount balancing would quite obviously work ($0 + 0 + 0 = 0$).[4] However, input and output groupings could just as trivially be identified based on testing if input/output subsets have balancing amounts.[5]

We can easily get around this by computing shared secrets between each pair of participants, then adding these offsets to their pseudo output commitments' masks (Section 5.4). In each pair, one participant adds the shared secret to one of his pseudo output commitments, and the other participant subtracts it from one of *his* pseudo commitments. When summed together, the secrets cancel out, and since each pair of participants has a shared secret the amount balance only appears after all commitments are combined.[6]

Shared secrets may hide input/output groupings in the immediate sense, but participants must learn about all the inputs and outputs somehow, and the easiest way is if they each communicate their individual input/output groupings. Clearly this violates the initial premise, and in any case implies participants know the number of participants.

11.1.1 n-way communication channel

The maximum number of participants to a TxTangle is either the number of outputs or inputs (whichever is lower). We model that by each real participant pretending to be a different person for each output he is sending. This is primarily for the purpose of setting up a group communication channel with other would-be participants, without revealing how many participants there are.

Imagine n ($2 \leq n \leq 16$, although at least 3 is recommended)[7] supposedly unrelated people gather at apparently random intervals in a chatroom, scheduled to open at time t_0 and close at t_1 (only 16 people can be in a chatroom at once, and the chatroom has conditions like fee priority, base fee per byte [to simplify consensus around the current median and block reward], and range of acceptable transaction types since e.g. currently transactions from the beginning of Monero can't be directly spent in a RingCT transaction [132]). At t_1 all mock-members signal desire to proceed

[4] Since Bulletproofs-style range proofs are actually aggregated into one (Section 5.5), participants would have to collaborate to some extent even in the trivial case.

[5] Participants could try to divide up the fee in a fancy manner to confuse observers, but it would fail in the face of brute force since fees are not that large (around 32 bits or less).

[6] We offset the pseudo output commitments instead of output commitments since output commitment masks are constructed from the recipient's address (Section 5.3).

[7] Currently, a transaction may have at most 16 outputs.

by publishing a public key, and the room is converted into an n-way communication channel by constructing a shared secret amongst all the mock-members.[8] This shared secret is used to encrypt message contents, while mock-members sign input-related messages using SAG signatures (Section 3.3) so it's never clear who sent a given message, and output-related messages with a bLSAG (Section 3.4) on the set of mock-member public keys so actual outputs are dissociated from mock-members.[9]

11.1.2 Message rounds to construct a joint transaction

After the channel is set up, TxTangle transactions can be constructed in five rounds of communication, where the next round may only begin after the previous is finished, and each round is given a timed communication interval within which messages should be randomly published. These intervals are intended to prevent message clusters that would reveal input/output groupings.

1. Each mock-member privately generates a random scalar for each intended output, and signs them with bLSAGs. A sorted list of these scalars is used to determine output indices (recall Section 4.2.1; the smallest scalar gets index $t = 0$).[10] They publish those bLSAGs, and also SAGs which sign the transaction version numbers of intended inputs. After this round participants can calculate the transaction weight based on number of inputs and outputs, and accurately estimate the fee required.[11,12,13]

2. Each mock-member uses the list of public keys to construct a shared secret with each other member for offsetting their pseudo output commitments, and decides who will add or subtract based on each pair's smaller public key.[14] Each mock-member must pay for $1/n^{\text{th}}$ of the fee estimate (using integer division). The mock-member with lowest output index is given the responsibility of paying for the remainder after dividing (it will be a truly infinitesimal amount, but must be taken into account to prevent fingerprinting TxTangle transactions). They privately generate transaction public keys for each of their outputs (not to be sent to

[8] The multisig method from Section 9.6.3 is one way, extending M-of-N all the way to 1-of-N.

[9] Each separate set of TxTangle bLSAGs should use the same key images, since everything related to a given output is linked together.

[10] Output index selection should match other implementations of transaction construction to avoid fingerprinting different software. We use this effectively random approach to align with the core implementation, which also randomizes outputs.

[11] If it turns out there must be only two real participants, based on comparing the number of inputs and outputs to one's own input/output count, the TxTangle can be abandoned. It's recommended for each participant to have at least two inputs and two outputs, in case of malicious actors who don't abandon TxTangles even when they realize it's just two participants. This recommendation is open to debate, since using more inputs and outputs is not heuristically neutral.

[12] TxTangle transactions should not have extraneous information stored in the extra field (e.g. no encrypted payment ID unless it's only a 2-output TxTangle which should have at least a dummy encrypted payment ID).

[13] Fee estimation should be based on a standardized approach, so each participant calculates the same thing. Otherwise outputs may be clustered based on fee calculation method. This same fee standard should be implemented outside of TxTangle, to promote TxTangle transactions looking the same as normal transactions.

[14] Since points are compressed (Section 2.4.2), just interpret the keys as 32-byte integers. The smaller key's owner adds, and larger key's owner subtracts, by convention.

other members just yet), and construct their output commitments, encoded amounts, and Part A partial proofs to be used for the aggregate Bulletproof range proof, signing all of it with bLSAGs (one commitment, one encoded amount, and one partial proof per bLSAG message, and the key image links these to the original list of random scalars that was used to specify output indices). Pseudo output commitments are generated as normal (Section 5.4), then offset with the shared secrets, and signed with SAGs. After the bLSAGs and SAGs are published, and assuming participants estimated the total fee in the same way, they may now verify that amounts balance overall.[15]

3. If amounts balance properly we begin a small additional round for building the aggregate Bulletproof that proves all output amounts are in range. Each mock-member uses the previous round's Part A partial proofs and output commitments, and privately computes the aggregate challenge A. They use it to construct their Part B partial proof, which they send to the channel with a bLSAG.

4. Participants begin filling in the message to be signed by MLSAGs (recall the footnote in Section 6.2.2). Published in random order over the communication interval are two kinds of messages. Each input's ring member offsets and key images are signed with a SAG and associated with the correct pseudo output commitment. Each output's one-time address, transaction public key, and Part C partial proof (computed based on the Part B partial proofs and an aggregate challenge B) are signed with a bLSAG (these can also include a random base transaction public key component, which as we will see can be used for fake Janus mitigation).

5. Participants use all the partial proofs to complete the aggregate Bulletproof, and privately apply a logarithmic inner product technique to compress it for the final proof to be included in transaction data. Once all the information to be signed by MLSAGs is collected, each participant completes their inputs' MSLAGs and randomly sends them (with a SAG for each) to the channel over the communication interval. Any participant may submit the transaction as soon as they have all the pieces.

Transaction public keys and the Janus mitigation

If every participant to a TxTangle knows the transaction private key r (Section 4.2), then any of them may test the others' one-time output addresses against a list of known addresses. For this reason it is necessary to construct TxTangle transactions as if there were a subaddress recipient (Section 4.3), including different transaction public keys for each output.

To match with possible implementation of a mitigation for the Janus attack related to subaddresses, in which one additional 'base' transaction public key is included in the extra field [101], TxTangles

[15] We don't publish the separate fee amounts paid for in case a participant calculated it wrong, which may reveal an output cluster due to a collection of non-standard fee amounts. If amounts don't balance properly, the TxTangle transaction may be abandoned.

should also have a fake 'base' key composed of a sum of random keys generated by each mock-member.[16]

Many TxTangle participants sending money to a subaddress will likely have at least two outputs, one of which diverts change back to the participant. This means any TxTangle participant can still enable Janus mitigation by making their change's transaction public key also the 'base' key for the subaddress recipient.[17] The subaddress recipient might realize the transaction is a TxTangle, and that the 'base' key probably corresponds with the sender's change output.[18]

11.1.3 Weaknesses

Malicious actors have two primary ways to defeat the purpose of TxTangle, which is to hide input/output groupings from potential adversaries/analysts. They may pollute the transactions, such that the subset of honest participants is smaller (or even non-existent) [91]. They may also cause TxTangle attempts to fail, and use subsequent attempts by the same participants to estimate input/output groupings.

The former case is not easy to mitigate, especially in the very decentralized case where no participant has a reputation. One possible application of TxTangle is with collaborating pools, who may hide which pool their miners belong to among a collection of pools. Such pools would know the input/output groupings, but since the purpose is helping their connected miners it would behoove them to keep the information secret. Moreover, such TxTangles would not permit bad actors, assuming the pools are somewhat honest.

The latter case can be defended against by only attempting to TxTangle a few times before taking a break, and by always regenerating most random elements of a transaction for new attempts. These elements include the transaction public keys, pseudo commitment masks, range proof scalars, and MLSAG scalars. In particular, the set of ring decoys for each input should remain the same to prevent cross-comparisons revealing the true input. If possible, different true inputs should be used for different TxTangle attempts. Since this weakness is inevitable, it makes the next section's concept more palatable.

[16] Key cancellation (Section 9.2.2) should not be a problem, since it's just a fake key and should ideally be randomly indexed within the list of transaction public keys.

[17] If sending to your own subaddress, there is no need for Janus mitigation. Wallets enabled for Janus mitigation should recognize the amount spent in a TxTangle equals the amount received to your subaddress, so they don't erroneously notify the user of a problem.

[18] This is assuming transaction public keys are 1:1 with the outputs, as is apparently the case today. If it was standard for transaction public keys to be in random or sorted order within the extra field, then TxTangle and non-TxTangle transactions would be largely indistinguishable for subaddress recipients. There are niche cases where TxTangle participants are unable to include a 'base' key (e.g. when all their outputs are to subaddresses), or where it is clearly non-TxTangle since the subaddress recipient receives most or all of the outputs. Note that since TxTangle transactions would generally have a lot more outputs than a typical transaction, this heuristic can be used to differentiate TxTangles from normal subaddress'd transactions.

11.2 Hosted TxTangle

Truly decentralized TxTangle has some open questions. How are the timed rounds initiated and enforced? How are chatrooms created in the first place, for participants to find each other? The most straightforward way is with a TxTangle host, who generates and manages those chatrooms.

Such a host would seem to defy the goal of obfuscated participation, since each individual must connect, and send it messages that could be used to correlate input/output groupings (especially if the host participates and knows the message contents). We can use a network like I2P[19] to make each message received by the host appear as if from a unique individual.

11.2.1 Basic communication with a host over I2P, and other features

With I2P, users make so-called 'tunnels' that pass heavily encrypted messages through other users' clients before reaching their destination. From what we understand, these tunnels can transport multiple messages before being destroyed and recreated (e.g. there appears to be a 10-minute timer on tunnels). It is essential for our use case to carefully control when new tunnels are created, and which messages may come out of the same tunnel.[20]

1. *Applying for TxTangles*: In our original n-way proposal (Section 11.1.1) participants gradually add their mock-members to available TxTangle rooms before they are scheduled to close. However, if a large enough volume of users try to TxTangle concurrently, there is likely to be a high rate of failure as users try to randomly put all their intended outputs in the same TxTangle 'room', but then the rooms get full too soon so they have to back out. It would be quite a mess.

   We can make an impactful optimization by telling the host how many outputs we have (e.g. giving him a list of our mock-member public keys), and letting him assemble each TxTangle's participants. Since we still retain the bLSAG and SAG messaging protocol, the host won't be able to identify output groupings in the final transaction. All he knows is the number of participants, and how many outputs each had. Moreover, in this scenario observers can't monitor open TxTangle rooms to deduce information about the participants, an important privacy improvement. Note that the host's power to pollute TxTangles isn't significantly different from the non-host design, so this change is neutral to that attack vector.

2. *Communication method*: Since the host is already acting as the locus of message transport, it is simplest for him to manage TxTangle communication. During each round the host collects messages from mock-members (still at random over a communication interval), and

[19] The Invisible Internet Project (https://geti2p.net/en/).

[20] In I2P there are 'outbound' and 'inbound' tunnels (see https://geti2p.net/en/docs/how/tunnel-routing). Everything received through an inbound tunnel looks like it's from the same source even if from multiple sources, so on the surface it would appear TxTangle users don't need to create different tunnels for all their usecases. However, if the TxTangle host makes himself the entry point for his own inbound tunnel, then he gains direct access to the outbound tunnels of TxTangle participants.

at the end of a round there is a short data distribution phase where he sends all the collected data to each participant, with a buffer period before the next round to ensure the messages are received and given time to process.

3. *Tunnels and input/output groupings*: Once a TxTangle has been initiated, users should dissociate their mock-member identities from the actual outputs that get created. This means creating new tunnels for bLSAG-signed messages, and each such tunnel may only transmit messages related to a specific output (it is fine to transmit multiple such messages through the same tunnel, since obviously information about the same output comes from the same source). They should also create new tunnels for SAG-signed messages related to specific inputs.

4. *Threat of host MITM attack*: The host might fool a participant by pretending to be other participants, since he controls sending out the mock-member list for constructing bLSAGs and SAGs. In other words, the list he sends participant A might contain participant A's mock-members, and all the rest are his own. Messages received by participant B are re-signed using A's list before being retransmitted to A. Since all messages signed by A's list belong to A, the host would have direct insight into A's input/output groupings!

   We can prevent the host from acting as MITM of honest participant interactions by modifying how transaction public keys are made. Participants send each other their intended transaction public keys as normal (with a bLSAG), then, much like robust key aggregation from Section 9.2.3, the actual keys that get included in transaction data (and used to make output commitment masks, etc.) are prefixed with a hash of the mock-member list. In other words, $\mathcal{H}_n(T_{agg}, \mathbb{S}_{mock}, r_t G) * r_t G$ is the t^{th} transaction public key. Baking the mock-member list into the transaction itself makes it very difficult to complete TxTangles without direct communication between all actual participants.[21]

11.2.2 Host as a service

It's important for sustainability and continuous improvement that a TxTangle service operate for profit.[22] Rather than compromise user identities with an account-based model, the host may participate in each TxTangle as a lone output, and require the participants to fund it. When accessing the host's 'eepsite'/service to apply for TxTangles, users are notified of the current hosting charge, which should be paid on a per-output basis.

Participants would be responsible for paying fractions of the fee *and* the host charge. This time, the smallest mock-member's public key (excluding the host's key) would take the remainder of both the fee and host charge.[23] Since the host has no inputs, he has no pseudo output commitment

[21] If Janus mitigation is implemented, this MITM defense should instead be done with the fake Janus base key. Each mock-member provides a random key $r_{mock}G$, then the actual base key is $\sum_{mock} \mathcal{H}_n(T_{agg}, \mathbb{S}_{mock}, r_{mock}G) * r_{mock}G$.

[22] While a deployed TxTangle service may be for profit, the code itself could be open source. This would be important for auditing wallet software that interacts with a TxTangle service.

[23] We must use mock-member keys here since the host doesn't pay a fee, and his output index is unknown.

to cancel his output's commitment mask. Instead, he creates shared secrets with the other mock-members as usual, then separates his real commitment mask into randomly sized chunks for each other mock-member and divides them by the shared secrets. He publishes a list of those scalars (corresponding them with each other mock-member based on their public key), signing with his mock-member key so participants know it's from the host. The appearance of this list signals the beginning of round 1 from Section 11.1.2 (e.g. the end of setup round '0'). Mock-members will multiply their host-scalar by the appropriate shared secret, and add that to their pseudo commitment mask. In this way, even the host's output cannot be identified by any participant in the final transaction without a full coalition against him.

To simplify calculations of the fee, the host may distribute the total fee to be used in the transaction at the end of round 1, since he will learn the transaction weight early. Participants can verify that amount is similar to the expected amount, and pay their fraction of it.

If participants collaborate to cheat, and not provide the hosting charge, then the host may terminate the TxTangle at round 3. He may also terminate if messages appear in the channel that should not be there or that aren't valid.

At the end of round 5 the host completes the transaction and submits it to the network for verification, as part of his service. He includes the transaction hash in the final distribution message.

11.3 Using a trusted dealer

There are some drawbacks to decentralized TxTangle. It requires all participants actively communicate within a strict timing schedule (and find each other to begin with), and is challenging to implement.

Adding a central dealer, who is responsible for collecting transaction information from each participant and obfuscating the input/output groupings, can simplify the proceedings. The cost is a higher bar of trust, since the dealer must (at minimum) learn those groupings.[24]

11.3.1 Dealer-based procedure

The dealer will advertise that he is available to manage TxTangles, and collects applications from potential participants (consisting of the number of intended inputs [with their types] and outputs). He may participate with his own input/output set if he wishes.

After a grouping of almost 16 outputs is assembled (there should be two or more participants, and no participant may have all but one output or input), the dealer will start the first of five rounds. In each round he accumulates information from each participant, makes some decisions, and sends out messages that signify the beginning of a new round.

[24] This section is inspired by the MoJoin protocol outline.

1. To begin, the dealer generates, for each pair of participants, a random scalar, and decides which in each pair should have the positive or negative version. He uses the number and type of inputs and outputs to estimate the total fee required. He sums each participant's scalars together, and privately sends to each their sum, along with the fee fraction they are expected to pay for, and the (randomly chosen) indices of their outputs. These messages constitute a signal to participants that a TxTangle is beginning.

2. Each participant constructs their sub-transaction as they normally would, generating separate transaction public keys for their outputs (with Janus mitigation as needed), calculating one-time output addresses and encoding output amounts, making pseudo output commitments that balance output commitments and the fee fraction, assembling a list of ring member offsets for use in MLSAG signatures along with the appropriate key images, and add to one of their pseudo output commitments the dealer-sent scalar (multiplied by G). They create Part A partial proofs for their outputs, and send all this information to the dealer. The dealer verifies that input and output amounts balance, and sends the complete list of Part A partial proofs to each participant.

3. Each participant computes the aggregate challenge A, and generates Part B partial proofs which they send to the dealer. The dealer collects the partial proofs and distributes them to all the other participants.

4. Each participant computes the aggregate challenge B, and generates Part C partial proofs which they send to the dealer. The dealer collects these, and applies the logarithmic inner product technique to compress them into the final proof. Assuming the proof verifies as it should, he generates a random fake Janus 'base' transaction public key, and sends the message to be signed in MLSAGs to each participant.

5. Each participant completes their MLSAGs and sends them to the dealer. Once he has all the pieces, he may finish constructing the transaction, and submit it to be included in the blockchain. He may also send the transaction ID to each participant so they can confirm it was published.

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] CoinJoin. https://en.bitcoin.it/wiki/CoinJoin [Online; accessed 01/26/2020].

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

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

[5] Cryptonote Address Tests. https://xmr.llcoins.net/addresstests.html [Online; accessed 04/19/2018].

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

[7] Extended Euclidean algorithm. https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm [Online; accessed 03/19/2020].

[8] hackerone #501585. https://hackerone.com/reports/501585 [Online; accessed 12/28/2019].

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

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

[11] Monero 0.13.0 "Beryllium Bullet" Release. https://www.getmonero.org/2018/10/11/monero-0.13.0-released.html [Online; accessed 10/12/2019].

[12] Monero 0.14.0 "Boron Butterfly" Release. https://web.getmonero.org/2019/02/25/monero-0.14.0-released.html [Online; accessed 10/12/2019].

[13] 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].

[14] 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].

[15] Randomx. https://www.monerooutreach.org/stories/RandomX.php [Online; accessed 10/12/2019].

[16] Ring CT; Moneropedia. https://getmonero.org/resources/moneropedia/ringCT.html [Online; accessed 06/05/2018].

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

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

[19] 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].

[20] What is a premine? https://www.cryptocompare.com/coins/guides/what-is-a-premine/ [Online; accessed 06/11/2018].

[21] 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].

[22] XOR – from Wolfram Mathworld. http://mathworld.wolfram.com/XOR.html [Online; accessed 04/21/2018].

[23] Federal Information Processing Standards Publication (FIPS 197). Advanced Encryption Standard (AES), 2001. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf [Online; access 03/04/2020].

[24] The .xz File Format, August 2009. https://tukaani.org/xz/xz-file-format.txt section 1.2 [Online; accessed 04/02/2020].

[25] Fungible, July 2014. https://wiki.mises.org/wiki/Fungible [Online; accessed 03/31/2020].

[26] Analysis of Bitcoin Transaction Size Trends, October 2015. https://tradeblock.com/blog/analysis-of-bitcoin-transaction-size-trends [Online; accessed 01/17/2020].

[27] 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].

[28] Base58Check encoding, November 2017. https://en.bitcoin.it/wiki/Base58Check_encoding [Online; accessed 02/20/2020].

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

[30] Payment Reversal Explained + 10 Ways to Avoid Them, October 2018. https://tidalcommerce.com/learn/payment-reversal [Online; accessed 02/11/2020].

[31] Diffie–Hellman problem, December 2019. https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_problem [Online; accessed 03/31/2020].

[32] Kurt M. Alonso and Jordi Herrera Joancomartí. Monero - Privacy in the Blockchain. Cryptology ePrint Archive, Report 2018/535, 2018. https://eprint.iacr.org/2018/535.

[33] Kurt M. Alonso and Koe. Zero to Monero - First Edition, June 2018. https://web.getmonero.org/library/Zero-to-Monero-1-0-0.pdf [Online; accessed 01/15/2020].

[34] Kristov Atlas. Kristov Atlas Security Advisory 20140609-0, June 2014. https://www.coinjoinsudoku.com/advisory/ [Online; accessed 01/26/2020].

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

[36] Daniel J. Bernstein. Curve25519: New Diffie-Hellman Speed Records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography - PKC 2006*, pages 207–228, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. https://cr.yp.to/ecdh/curve25519-20060209.pdf [Online; accessed 03/04/2020].

[37] Daniel J. Bernstein. ChaCha, a variant of Salsa20, January 2008. https://cr.yp.to/chacha/chacha-20080120.pdf [Online; accessed 03/04/2020].

[38] Daniel J. Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. *Twisted Edwards Curves*, pages 389–405. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. https://eprint.iacr.org/2008/013.pdf [Online; accessed 02/13/2020].

[39] 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. https://ed25519.cr.yp.to/ed25519-20110705.pdf [Online; accessed 03/04/2020].

[40] Daniel J. Bernstein and Tanja Lange. *Faster Addition and Doubling on Elliptic Curves*, pages 29–50. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. https://eprint.iacr.org/2007/286.pdf [Online; accessed 03/04/2020].

[41] 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].

[42] Dan Boneh and Victor Shoup. A Graduate Course in Applied Cryptography. https://crypto.stanford.edu/~dabo/cryptobook/ [Online; accessed 12/30/2019].

[43] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short Proofs for Confidential Transactions and More. https://eprint.iacr.org/2017/1066.pdf [Online; accessed 10/28/2018].

[44] Francisco Cabañas. Francisco Cabanas - Critical Role of Min Block Reward Trail Emission - DEF CON 27 Monero Village, December 2019. https://www.youtube.com/watch?v=IlghysBBuyU [Online; accessed 01/15/2020].

[45] Francisco Cabañas. Lightning talk: An Overview of Monero's Adaptive Blockweight Approach to Scaling, December 2019. https://frab.riat.at/en/36C3/public/events/125.html [Online; accessed 01/12/2020].

[46] Francisco Cabañas. MoneroKon 2019 - Spam Mitigation and Size Control in Permissionless Blockchains, June 2019. https://www.youtube.com/watch?v=Hbm0ub3qWw4&list=LL2HXH-vq_sTPXMNP4fZNKRw&index=10&t=0s [Online; accessed 01/10/2020].

[47] Miles Carlsten, Harry Kalodner, Matthew Weinberg, and Arvind Narayanan. On the Instability of Bitcoin Without the Block Reward, 2016. http://randomwalker.info/publications/mining_CCS.pdf [Online; accessed 01/12/2020].

[48] Chainalysis. THE 2020 STATE OF CRYPTO CRIME, January 2020. https://go.chainalysis.com/rs/503-FAP-074/images/2020-Crypto-Crime-Report.pdf [Online; accessed 02/11/2020].

[49] 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. https://chaum.com/publications/Group_Signatures.pdf [Online; accessed 03/04/2020].

[50] dalek cryptography. Bulletproofs. https://doc-internal.dalek.rs/bulletproofs/index.html [Online; accessed 03/02/2020].

[51] Michael Davidson and Tyler Diamond. On the Profitability of Selfish Mining Against Multiple Difficulty Adjustment Algorithms. Cryptology ePrint Archive, Report 2020/094, 2020. https://eprint.iacr.org/2020/094 [Online; accessed 03/26/2020].

[52] W. Diffie and M. Hellman. New Directions in Cryptography. *IEEE Trans. Inf. Theor.*, 22(6):644–654, September 2006. https://ee.stanford.edu/~hellman/publications/24.pdf [Online; accessed 03/04/2020].

[53] Manu Drijvers, Kasra Edalatnejad, Bryan Ford, Eike Kiltz, Julian Loss, Gregory Neven, and Igors Stepanovs. On the Security of Two-Round Multi-Signatures. Cryptology ePrint Archive, Report 2018/417, 2018. https://eprint.iacr.org/2018/417 [Online; accessed 02/07/2020].

[54] Justin Ehrenhofer. Justin Ehrenhofer - Improving Monero Release Schedule - DEF CON 27 Monero Village, December 2019. https://www.youtube.com/watch?v=MjNXmJUk2Jo timestamp 22:15 [Online; accessed 03/20/2020].

[55] Justin Ehrenhofer. Monero Adds Blockchain Pruning and Improves Transaction Efficiency, February 2019. https://web.getmonero.org/2019/02/01/pruning.html [Online; accessed 12/30/2019].

[56] Justin Ehrenhofer and knacc. Advisory note for users making use of subaddresses, October 2019. https://web.getmonero.org/2019/10/18/subaddress-janus.html [Online; accessed 01/02/2020].

[57] Serhack et al. Mastering Monero, December 2019. https://masteringmonero.com/ [Online; accessed 01/10/2020].

[58] Exantech. Methods of anonymous blockchain analysis: an overview, November 2019. https://medium.com/@exantech/methods-of-anonymous-blockchain-analysis-an-overview-d700e27ea98c [Online; accessed 01/26/2020].

[59] Ittay Eyal and Emin Gün Sirer. Majority is not Enough: Bitcoin Mining is Vulnerable. *CoRR*, abs/1311.0243, 2013. https://arxiv.org/pdf/1311.0243.pdf [Online; accessed 03/04/2020].

[60] 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. https://link.springer.com/content/pdf/10.1007%2F3-540-47721-7_12.pdf [Online; accessed 03/04/2020].

[61] Ryo "fireice_uk" Cryptocurrency. On-chain tracking of Monero and other Cryptonotes, April 2019. https://medium.com/@crypto_ryo/on-chain-tracking-of-monero-and-other-cryptonotes-e0afc6752527 [Online; accessed 03/25/2020].

[62] Riccardo "fluffypony" Spagni. Monero 0.12.0.0 "Lithium Luna" Release, March 2018. https://web.getmonero.org/2018/03/29/monero-0.12.0.0-released.html [Online; accessed 02/18/2020].

[63] Riccardo "fluffypony" Spagni and luigi1111. 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].

[64] David Friedman. A Positive Account of Property Rights, 1994. http://www.daviddfriedman.com/Academic/Property/Property.html [Online; accessed 03/18/2020].

[65] Eiichiro Fujisaki and Koutarou Suzuki. *Traceable Ring Signature*, pages 181–200. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. https://link.springer.com/content/pdf/10.1007%2F978-3-540-71677-8_13.pdf [Online; accessed 03/04/2020].

[66] glv2. Varint description; Issue #2340. https://github.com/monero-project/monero/issues/2340#issuecomment-324692291 [Online; accessed 06/14/2018].

[67] Brandon Goodell, Sarang Noether, and Arthur Blue. Concise linkable ring signatures and applications, MRL-0011, September 2019. https://web.getmonero.org/resources/research-lab/pubs/MRL-0011.pdf [Online; accessed 02/02/2020].

[68] Thomas C Hales. The NSA back door to NIST. *Notices of the AMS*, 61(2):190–192. https://www.ams.org/notices/201402/rnoti-p190.pdf [Online; accessed 03/04/2020].

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

[70] Howard "hyc" Chu. RandomX, Pull Request #5549, May 2019. https://github.com/monero-project/monero/pull/5549 [Online; accessed 03/03/2020].

[71] Aram Jivanyan. Lelantus: Towards Confidentiality and Anonymity of Blockchain Transactions from Standard Assumptions. Cryptology ePrint Archive, Report 2019/373, 2019. https://eprint.iacr.org/2019/373.pdf [Online; accessed 03/04/2020].

[72] 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].

[73] JollyMort. Monero Dynamic Block Size and Dynamic Minimum Fee, March 2017. https://github.com/JollyMort/monero-research/blob/master/Monero%20Dynamic%20Block%20Size%20and%20Dynamic%20Minimum%20Fee/Monero%20Dynamic%20Block%20Size%20and%20Dynamic%20Minimum%20Fee%20-%20DRAFT.md [Online; accessed 01/12/2020].

[74] S. Josefsson, SJD AB, 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].

[75] Simon Josefsson and Ilari Liusvaara. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032, January 2017. https://rfc-editor.org/rfc/rfc8032.txt [Online; accessed 03/04/2020].

[76] kenshi84. Subaddresses, Pull Request #2056, May 2017. https://github.com/monero-project/monero/pull/2056 [Online; accessed 02/16/2020].

[77] Eike Kiltz, Daniel Masny, and Jiaxin Pan. Optimal Security Proofs for Signatures from Identification Schemes. In *Proceedings, Part II, of the 36th Annual International Cryptology Conference on Advances in Cryptology — CRYPTO 2016 - Volume 9815*, pages 33–61, Berlin, Heidelberg, 2016. Springer-Verlag. https://eprint.iacr.org/2016/191.pdf [Online; accessed 03/04/2020].

[78] Bradley Kjell. Big Endian and Little Endian. https://chortle.ccsu.edu/AssemblyTutorial/Chapter-15/ass15_3.html [Online; accessed 01/23/2020].

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

[80] koe and jtgrassie. Historical significance of FEE_PER_KB_OLD, December 2019. https://monero.stackexchange.com/questions/11864/historical-significance-of-fee-per-kb-old [Online; accessed 01/02/2020].

[81] koe and jtgrassie. Complete extra field structure (standard interpretation), January 2020. https://monero.stackexchange.com/questions/11888/complete-extra-field-structure-standard-interpretation [Online; accessed 01/05/2020].

[82] Mitchell Krawiec-Thayer. MoneroKon 2019 - Visualizing Monero: A Figure is Worth a Thousand Logs, June 2019. https://www.youtube.com/watch?v=XIrqyxU3k5Q [Online; accessed 01/06/2020].

[83] Mitchell Krawiec-Thayer. Numerical simulation for upper bound on dynamic blocksize expansion, January 2019. https://github.com/noncesense-research-lab/Blockchain_big_bang/blob/master/models/Isthmus_Bx_big_bang_model.ipynb [Online; accessed 01/08/2020].

[84] Russell W. F. Lai, Viktoria Ronge, Tim Ruffing, Dominique Schröder, Sri Thyagarajan, and Jiafan Wang. Omniring: Scaling Private Payments Without Trusted Setup. pages 31–48, 11 2019. https://eprint.iacr.org/2019/580.pdf [Online; accessed 03/04/2020].

[85] 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. https://eprint.iacr.org/2004/027.pdf [Online; accessed 03/04/2020].

[86] Jan Macheta, Sarang Noether, Surae Noether, and Javier Smooth. Counterfeiting via Merkle Tree Exploits within Virtual Currencies Employing the CryptoNote Protocol, MRL-0002, September 2014. https://web.getmonero.org/resources/research-lab/pubs/MRL-0002.pdf [Online; accessed 05/27/2018].

[87] 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. https://www.crypto.ethz.ch/publications/files/Maurer09.pdf [Online; accessed 03/04/2020].

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

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

[90] Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille. Simple Schnorr Multi-Signatures with Applications to Bitcoin. May 2018. https://eprint.iacr.org/2018/068.pdf [Online; accessed 03/01/2020].

[91] Sarah Meiklejohn and Claudio Orlandi. Privacy-Enhancing Overlays in Bitcoin. https://fc15.ifca.ai/preproceedings/bitcoin/paper_5.pdf [Online; accessed 01/26/2020].

[92] R. C. Merkle. Protocols for Public Key Cryptosystems. In *1980 IEEE Symposium on Security and Privacy*, pages 122–122, April 1980. http://www.merkle.com/papers/Protocols.pdf [Online; accessed 03/04/2020].

[93] Andrew Miller, Malte Möser, Kevin Lee, and Arvind Narayanan. An Empirical Analysis of Linkability in the Monero Blockchain. *CoRR*, abs/1704.04299, 2017. https://arxiv.org/pdf/1704.04299.pdf [Online; accessed 03/04/2020.

[94] 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. https://link.springer.com/content/pdf/10.1007/3-540-39799-X_31.pdf [Online; accessed 03/04/2020].

[95] Nicola Minichiello. The Bitcoin Big Bang: Tracking Tainted Bitcoins, June 2015. https://bravenewcoin.com/insights/the-bitcoin-big-bang-tracking-tainted-bitcoins [Online; accessed 03/31/2020].

[96] Ludwig Von Mises. Human Action: A Treatise on Economics; The Scholar's Edition, 1949. https://cdn.mises.org/Human%20Action_3.pdf [Online; accessed 03/05/2020].

[97] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008. http://bitcoin.org/bitcoin.pdf [Online; accessed 03/04/2020].

[98] Arvind Narayanan. Bitcoin is unstable without the block reward, October 2016. https://freedom-to-tinker.com/2016/10/21/bitcoin-is-unstable-without-the-block-reward/ [Online; accessed 01/12/2020].

[99] Arvind Narayanan and Malte Möser. Obfuscation in Bitcoin: Techniques and Politics. *CoRR*, abs/1706.05432, 2017. https://arxiv.org/ftp/arxiv/papers/1706/1706.05432.pdf [Online; accessed 03/04/2020].

[100] Y. Nir, Check Point, A. Langley, and Google Inc. ChaCha20 and Poly1305 for IETF Protocols. Internet Research Task Force (IRTF), May 2015. https://tools.ietf.org/html/rfc7539 [Online; accessed 05/11/2018].

[101] Sarang Noether. Janus mitigation, Issue #62, January 2020. https://github.com/monero-project/research-lab/issues/62 [Online; accessed 02/17/2020].

[102] Sarang Noether. Multisignature implementation, Issue #67, January 2020. https://github.com/monero-project/research-lab/issues/67 [Online; accessed 02/16/2020].

[103] Sarang Noether. Transaction proofs (InProofV1 and OutProofV1) have incomplete Schnorr challenges, Issue #60, January 2020. https://github.com/monero-project/research-lab/issues/60 [Online; accessed 02/20/2020].

[104] Sarang Noether. WIP: Updated transaction proofs and tests, Pull Request #6329, February 2020. https://github.com/monero-project/monero/pull/6329 [Online; accessed 02/20/2020].

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

[106] Sarang Noether and Brandon Goodell. Triptych: logarithmic-sized linkable ring signatures with applications. Cryptology ePrint Archive, Report 2020/018, 2020. https://eprint.iacr.org/2020/018.pdf [Online; accessed 03/04/2020.

[107] Shen Noether. Understanding ge_fromfe_frombytes_vartime. https://web.getmonero.org/resources/research-lab/pubs/ge_fromfe.pdf [Online; accessed 01/20/2020].

[108] Shen Noether, Adam Mackenzie, and Monero Core Team. Ring Confidential Transactions, MRL-0005, February 2016. https://web.getmonero.org/resources/research-lab/pubs/MRL-0005.pdf [Online; accessed 06/15/2018].

[109] Shen Noether, Adam Mackenzie, and Monero Core Team. Ring Multisignature, April 2016. https://web.archive.org/web/20161023010706/https://shnoe.files.wordpress.com/2016/03/mrl-0008_april28.pdf [Online; accessed via WayBack Machine 01/21/2020].

[110] Shen Noether and Sarang Noether. Monero is Not That Mysterious, MRL-0003, September 2014. https://web.getmonero.org/resources/research-lab/pubs/MRL-0003.pdf [Online; accessed 06/15/2018].

[111] Shen Noether and Sarang Noether. Thring Signatures and their Applications to Spender-Ambiguous Digital Currencies, MRL-0009, November 2018. https://web.getmonero.org/resources/research-lab/pubs/MRL-0009.pdf [Online; accessed 01/15/2020].

[112] 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].

[113] Torben Pryds Pedersen. *Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing*, pages 129–140. Springer Berlin Heidelberg, Berlin, Heidelberg, 1992. https://www.cs.cornell.edu/courses/cs754/2001fa/129.PDF [Online; accessed 03/04/2020].

[114] rbrunner7. 2/2 Multisig in CLI Wallet, January 2018. https://taiga.getmonero.org/project/rbrunner7-really-simple-multisig-transactions/wiki/22-multisig-in-cli-wallet [Online; accessed 01/21/2020].

[115] René "rbrunner7" Brunner. Multisig transactions with MMS and CLI wallet. https://web.getmonero.org/resources/user-guides/multisig-messaging-system.html [Online; accessed 01/21/2020].

[116] René "rbrunner7" Brunner. Project Rationale From the Initiator, January 2018. https://taiga.getmonero.org/project/rbrunner7-really-simple-multisig-transactions/wiki/home [Online; accessed 01/21/2020].

[117] René "rbrunner7" Brunner. Basic Monero support ready for assessment, Issue #1638, June 2019. https://github.com/OpenBazaar/openbazaar-go/issues/1638 [Online; accessed 02/11/2020].

[118] Jamie Redman. Industry Execs Claim Freshly Minted 'Virgin Bitcoins' Fetch 20% Premium, March 2020. https://news.bitcoin.com/industry-execs-freshly-minted-virgin-bitcoins/ [Online; accessed 03/31/2020].

[119] Ronald L. Rivest, Adi Shamir, 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].

[120] SafeCurves. SafeCurves: choosing safe curves for elliptic-curve cryptography, 2013. https://safecurves.cr.yp.to/rigid.html [Online; accessed 03/25/2020].

[121] 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. https://link.springer.com/content/pdf/10.1007%2F0-387-34805-0_22.pdf [Online; accessed 03/04/2020].

[122] Paola Scozzafava. Uniform distribution and sum modulo m of independent random variables. *Statistics & Probability Letters*, 18(4):313 – 314, 1993. https://sci-hub.tw/https://www.sciencedirect.com/science/article/abs/pii/016771529390021A [Online; accessed 03/04/2020].

[123] Bassam El Khoury Seguias. Monero Building Blocks, 2018. https://delfr.com/category/monero/ [Online; accessed 10/28/2018].

[124] 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].

[125] QingChun ShenTu and Jianping Yu. Research on Anonymization and De-anonymization in the Bitcoin System. *CoRR*, abs/1510.07782, 2015. https://arxiv.org/ftp/arxiv/papers/1510/1510.07782.pdf [Online; accessed 03/04/2020].

[126] stoffu. Reserve proof, Pull Request #3027, December 2017. https://github.com/monero-project/monero/pull/3027 [Online; accessed 02/24/2020].

[127] 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].

[128] Manny Trillo. Visa Transactions Hit Peak on Dec. 23, January 2011. https://www.visa.com/blogarchives/us/2011/01/12/visa-transactions-hit-peak-on-dec-23/index.html [Online; accessed 01/10/2020].

[129] Alicia Tuovila. Audit, July 2019. https://www.investopedia.com/terms/a/audit.asp [Online; accessed 02/24/2020].

[130] UkoeHB. Proof an output has not been spent, Issue #68, January 2020. https://github.com/monero-project/research-lab/issues/68 [Online; accessed 02/19/2020].

[131] UkoeHB. Reduce minimum fee variability, Issue #70, February 2020. https://github.com/monero-project/research-lab/issues/70 [Online; accessed 03/20/2020].

[132] UkoeHB. Treat pre-RingCT outputs like coinbase outputs, Issue #59, January 2020. https://github.com/monero-project/research-lab/issues/59 [Online; accessed 03/22/2020].

[133] Maciej Ulas. Rational points on certain hyperelliptic curves over finite fields, 2007. https://arxiv.org/pdf/0706.1448.pdf [Online; accessed 03/03/2020].

[134] user36100 and cooldude45. Which entities are related to Bytecoin and Minergate?, December 2016. https://monero.stackexchange.com/questions/2930/which-entities-are-related-to-bytecoin-and-minergate [Online; accessed 01/17/2020].

[135] user36303 and koe. Duplicate ring members, January 2020. https://monero.stackexchange.com/questions/11882/duplicate-ring-members [Online; accessed 01/18/2020].

[136] Nicolas van Saberhagen. CryptoNote V2.0. https://cryptonote.org/whitepaper.pdf [Online; accessed 04/04/2018].

[137] David Wagner. A Generalized Birthday Problem. In Moti Yung, editor, *Advances in Cryptology — CRYPTO 2002*, pages 288–304, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. https://link.springer.com/content/pdf/10.1007%2F3-540-45708-9_19.pdf [Online; accessed 02/07/2020].

[138] Adam "waxwing" Gibson. From Zero (Knowledge) To Bulletproofs, March 2018. https://github.com/AdamISZ/from0k2bp/blob/master/from0k2bp.pdf [Online; accessed 03/01/2020].

[139] Adam "waxwing" Gibson. Avoiding Wagnerian Tragedies, December 2019. https://joinmarket.me/blog/blog/avoiding-wagnerian-tragedies/ [Online; accessed 03/01/2020].

[140] Albert Werner, Montag, Prometheus, and Tereno. CryptoNote Transaction Extra Field. CryptoNote, October 2012. https://cryptonote.org/cns/cns005.txt [Online; accessed 04/04/2018].

[141] Ursula Whitcher. The Birthday Problem. http://mathforum.org/dr.math/faq/faq.birthdayprob.html [Online; accessed 02/07/2020].

[142] Wikibooks. Cryptography/Prime Curve/Standard Projective Coordinates, March 2011. https://en.wikibooks.org/wiki/Cryptography/Prime_Curve/Standard_Projective_Coordinates [Online; accessed 03/03/2020].

[143] Tsz Hon Yuen, Shi-feng Sun, Joseph K. Liu, Man Ho Au, Muhammed F. Esgin, Qingzhao Zhang, and Dawu Gu. RingCT 3.0 for Blockchain Confidential Transaction: Shorter Size and Stronger Security. Cryptology ePrint Archive, Report 2019/508, 2019. https://eprint.iacr.org/2019/508.pdf [Online; accessed 03/04/2020].

[144] zawy12. Summary of Difficulty Algorithms, Issue #50, December 2019. https://github.com/zawy12/difficulty-algorithms/issues/50 [Online; accessed 02/11/2020].

Appendices

APPENDIX A

`RCTTypeBulletproof2` Transaction Structure

We present in this appendix a dump from a real Monero transaction of type `RCTTypeBulletproof2`, together with explanatory notes for relevant fields.

The dump was obtained from the block explorer https://xmrchain.net. It can also be acquired by executing command `print_tx <TransactionID> +hex +json` in the `monerod` daemon run in non-detached mode. `<TransactionID>` is a hash of the transaction (Section 7.4.1). The first line printed shows the actual command to run.

To replicate our results, readers can do the following:

1. We need the Monero command line tool (CLI), which can be found at https://web.getmonero.org/downloads/ (among other places). Get the 'Command Line Tools Only' for your operating system, move the file to a useful location, and unzip it.
2. Open the terminal/command line and navigate into the folder created by unzipping.
3. Run `monerod` with `./monerod`. Now the Monero blockchain will download. Unfortunately, there is currently no easy way to print transactions on your own system (e.g. without using a block explorer) without downloading the blockchain.
4. After the syncing process is done, commands like `print_tx` will work. Use `help` to learn other commands.

Component `rctsig_prunable`, as indicated by its name, is *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 can be pruned and replaced with its hash for the Merkle tree.

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

Our sample transaction has 2 inputs and 2 outputs, and was added to the blockchain at timestamp 2020-03-02 19:01:10 UTC (as reported by its block's miner).

```
print_tx 84799c2fc4c18188102041a74cef79486181df96478b717e8703512c7f7f3349
Found in blockchain at height 2045821
{
  "version": 2,
  "unlock_time": 0,
  "vin": [ {
      "key": {
        "amount": 0,
        "key_offsets": [ 14401866, 142824, 615514, 18703, 5949, 22840, 5572, 16439,
        983, 4050, 320
        ],
        "k_image": "c439b9f0da76ca0bb17920ca1f1f3f1d216090751752b091bef9006918cb3db4"
      }
    }, {
      "key": {
        "amount": 0,
        "key_offsets": [ 14515357, 640505, 8794, 1246, 20300, 18577, 17108, 9824, 581,
        637, 1023
        ],
        "k_image": "03750c4b23e5be486e62608443151fa63992236910c41fa0c4a0a938bc6f5a37"
      }
    }
  ],
  "vout": [ {
      "amount": 0,
      "target": {
        "key": "d890ba9ebfa1b44d0bd945126ad29a29d8975e7247189e5076c19fa7e3a8cb00"
      }
    }, {
      "amount": 0,
      "target": {
        "key": "dbec330f8a67124860a9bfb86b66db18854986bd540e710365ad6079c8a1c7b0"
      }
```

```
      }
    ],
    "extra": [ 1, 3, 39, 58, 185, 169, 82, 229, 226, 22, 101, 230, 254, 20, 143,
    37, 139, 28, 114, 77, 160, 229, 250, 107, 73, 105, 64, 208, 154, 182, 158, 200,
    73, 2, 9, 1, 12, 76, 161, 40, 250, 50, 135, 231
    ],
    "rct_signatures": {
      "type": 4,
      "txnFee": 32460000,
      "ecdhInfo": [ {
          "amount": "171f967524e29632"
        }, {
          "amount": "5c2a1a9f54ccf40b"
        }],
      "outPk": [ "fed8aded6914f789b63c37f9d2eb5ee77149e1aa4700a482aea53f82177b3b41",
      "670e086e40511a279e0e4be89c9417b4767251c5a68b4fc3deb80fdae7269c17"]
    },
    "rctsig_prunable": {
      "nbp": 1,
      "bp": [ {
          "A": "98e5f23484e97bb5b2d453505db79caadf20dc2b69dd3f2b3dbf2a53ca280216",
          "S": "b791d4bc6a4d71de5a79673ed4a5487a184122321ede0b7341bc3fdc0915a796",
          "T1": "5d58cfa9b69ecdb2375647729e34e24ce5eb996b5275aa93f9871259f3a1aecd",
          "T2": "1101994fea209b71a2aa25586e429c4c0f440067e2b197469aa1a9a1512f84b7",
          "taux": "b0ad39da006404ccacee7f6d4658cf17e0f42419c284bdca03c0250303706c03",
          "mu": "cacd7ca5404afa28e7c39918d9f80b7fe5e572a92a10696186d029b4923fa200",
          "L": [ "d06404fc35a60c6c47a04e2e43435cb030267134847f7a49831a61f82307fc32",
          "c9a5932468839ee0cda1aa2815f156746d4dce79dab3013f4c9946fce6b69eff",
          "efdae043dcedb79512581480d80871c51e063fe9b7a5451829f7a7824bcc5a0b",
          "56fd2e74ac6e1766cfd56c8303a90c68165a6b0855fae1d5b51a2e035f333a1d",
          "81736ed768f57e7f8d440b4b18228d348dce1eca68f969e75fab458f44174c99",
          "695711950e076f54cf24ad4408d309c1873d0f4bf40c449ef28d577ba74dd86d",
          "4dc3147619a6c9401fec004652df290800069b776fe31b3c5cf98f64eb13ef2c"
          ],
          "R": [ "7650b8da45c705496c26136b4c1104a8da601ea761df8bba07f1249495d8f1ce",
          "e87789fbe99a44554871fcf811723ee350cba40276ad5f1696a62d91a4363fa6",
          "ebf663fe9bb580f0154d52ce2a6dae544e7f6fb2d3808531b0b0749f5152ddbf",
          "5a4152682a1e812b196a265a6ba02e3647a6bd456b7987adff288c5b0b556ec6",
          "dc0dcb2e696e11e4b62c20b6bfcb6182290748c5de254d64bf7f9e3c38fb46c9",
          "101e2271ced03b229b88228d74b36088b40c88f26db8b1f9935b85fb3ab96043",
          "b14aae1d35c9b176ac526c23f31b044559da75cf95bc640d1005bfcc6367040b"
          ],
```

```
            "a": "4809857de0bd6becdb64b85e9dfbf6085743a8496006b72ceb81e01080965003",
            "b": "791d8dc3a4ddde5ba2416546127eb194918839ced3dea7399f9c36a17f36150e",
            "t": "aace86a7a1cbdec3691859fa07fdc83eed9ca84b8a064ca3f0149e7d6b721c03"
          }
        ],
        "MGs": [ {
            "ss": [[ "d7a9b87cfa74ad5322eedd1bff4c4dca08bcff6f8578a29a8bc4ad6789dee106",
            "f08e5dfade29d2e60e981cb561d749ea96ddc7e6855f76f9b807842d1a17fe00"],
            ["de0a86d12be2426f605a5183446e3323275fe744f52fb439041ad2d59136ea0b",
            "0028f97976630406e12c54094cbbe23a23fe5098f43bcae37339bfc0c4c74c07"],
            ["f6eef1f99e605372cb1ec2b3dd4c6e56a550fec071c8b1d830b9fda34de5cc05",
            "cd98fc987374a0ac993edf4c9af0a6f2d5b054f2af601b612ea118f405303306"],
            ["5a8437575dae7e2183a1c620efbce655f3d6dc31e64c96276f04976243461e08",
            "5090103f7f73a33024fbda999cd841b99b87c45fa32c4097cdc222fa3d7e9502"],
            ["88d34246afbccbd24d2af2ba29d835813634e619912ea4ca194a32281ac14e0e",
            "eacdf59478f132dd8dbb9580546f96de194092558ffceeff410ee9eb30ce570e"],
            ["571dab8557921bbae30bda9b7e613c8a0cff378d1ec6413f59e4972f30f2470d",
            "5ca78da9a129619299304d9b03186233370023debfdaddcd49c1a338c1f0c50d"],
            ["ac8dbe6bb28839cf98f02908bd1451742a10c713fdd51319f2d42a58bf1d7507",
            "7347bf16cba5ee6a6f2d4f6a59d1ed0c1a43060c3a235531e7f1a75cd8c8530d"],
            ["b8876bd3a5766150f0fbc675ba9c774c2851c04afc4de0b17d3ac4b6de617402",
            "e39f1d2452d76521cbf02b85a6b626eeb5994f6f28ce5cf81adc0ff2b8adb907"],
            ["1309f8ead30b7be8d0c5932743b343ef6c0001cef3a4101eae98ffde53f46300",
            "370693fa86838984e9a7232bca42fd3d6c0c2119d44471d61eee5233ba53c20f"],
            ["80bc2da5fc5951f2c7406fce37a7aa72ffef9cfa21595b1b68dfab4b7b9f9f0c",
            "c37137898234f00bce00746b131790f3223f97960eefe67231eb001092f5510c"],
            ["01c89e07571fd365cac6744b34f1b44e06c1c31cbf3ee4156d08309345fdb20e",
            "a35c8786695a86c0a4e677b102197a11dadc7171dd8c2e1de90d828f050ec00f"]],
            "cc": "0d8b70c600c67714f3e9a0480f1ffc7477023c793752c1152d5df0813f75ff0f"
          }, {
            "ss": [[ "4536e585af58688b69d932ef3436947a13d2908755d1c644ca9d6a978f0f0206",
            "9aab6509f4650482529219a805ee09cd96bb439ee1766ced5d3877bf1518370b"],
            ["5849d6bf0f850fcee7acbef74bd7f02f77ecfaaa16a872f52479ebd27339760f",
            "96a9ec61486b04201313ac8687eaf281af59af9fd10cf450cb26e9dc8f1ce804"],
            ["7fe5dcc4d3eff02fca4fb4fa0a7299d212cd8cd43ec922d536f21f92c8f93f00",
            "d306a62831b49700ae9daad44fcd00c541b959da32c4049d5bdd49be28d96701"],
            ["2edb125a5670d30f6820c01b04b93dd8ff11f4d82d78e2379fe29d7a68d9c103",
            "753ac25628c0dada7779c6f3f13980dfc5b7518fb5855fd0e7274e3075a3410c"],
            ["264de632d9cb867e052f95007dfdf5a199975136c907f1d6ad73061938f49c01",
            "dd7eb6028d0695411f647058f75c42c67660f10e265c83d024c4199bed073d01"],
            ["b2ac07539336954f2e9b9cba298d4e1faa98e13e7039f7ae4234ac801641340f",
            "69e130422516b82b456927b64fe02732a3f12b5ee00c7786fe2a381325bf3004"],
```

```
118          ["49ea699ca8cf2656d69020492cdfa69815fb69145e8f922bb32e358c23cebb0f",
119          "c5706f903c04c7bed9c74844f8e24521b01bc07b8dbf597621cceeeb3afc1d0c"],
120          ["a1faf85aa942ba30b9f0511141fcab3218c00953d046680d36e09c35c04be905",
121          "7b6b1b6fb23e0ee5ea43c2498ea60f4fcf62f70c7e0e905eb4d9afa1d0a18800"],
122          ["785d0993a70f1c2f0ac33c1f7632d64e34dd730d1d8a2fb0606f5770ed633506",
123          "e12777c49ffc3f6c35d27a9ccb3d9b8fed7f0864a880f7bae7399e334207280e"],
124          ["ab31972bf1d2f904d6b0bf18f4664fa2b16a1fb2644cd4e6278b63ade87b6d09",
125          "1efb04fe9a75c01a0fe291d0ae00c716e18c64199c1716a086dd6e32f63e0a07"],
126          ["a6f4e21a27bf8d28fc81c873f63f8d78e017666adbf038da0b83c2ad04ef6805",
127          "c02103455f93c2d7ec4b7152db7de00d1c9e806b1945426b6773026b4a85dd03"]],
128          "cc": "d5ac037bb78db41cf924af713b7379c39a4e13901d3eac017238550a1a3b910a"
129        }],
130      "pseudoOuts": [ "b313c1ae9ca06213684fbdefa9412f4966ad192bc0b2f74ed1731381adb7ab58",
131      "7148e7ef5cfd156c62a6e285e5712f8ef123575499ff9a11f838289870522423"]
132    }
133  }
```

Transaction components

- (line 2) - The command `print_tx` would report the block where it found the transaction, which we replicate here for demonstration purposes.
- `version` (line 4) - Transaction format/era version; '2' corresponds to RingCT.
- `unlock_time` (line 5) - Prevents a transaction's 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. It defaults to zero when no limit is specified.
- `vin` (line 6-23) - List of inputs (there are two here)
- `amount` (line 8) - Deprecated (legacy) amount field for type 1 transactions
- `key_offset` (line 9) - This allows verifiers to find ring member keys and commitments in the blockchain, and makes it obvious those members are legitimate. The first offset is absolute within the blockchain history, and each subsequent offset is relative to the previous. For example, with real offsets {7,11,15,20}, the blockchain records {7,4,4,5}. Verifiers compute the last offset like (7+4+4+5 = 20) (Section 6.2.4).
- `k_image` (line 12) - Key image $\tilde{K_j}$ from Section 3.5, where $j = 1$ since this is the first input.
- `vout` (lines 24-35) - List of outputs (there are two here)
- `amount` (line 25) - Deprecated amount field for type 1 transactions
- `key` (line 27) - One-time destination key for output $t = 0$ as described in Section 4.2

- `extra` (lines 36-39) - Miscellaneous data, including the *transaction public key*, i.e. the share secret rG of Section 4.2, and encrypted payment ID from Section 4.4. It typically works like this: each number is one byte (it can be 0-255), and each kind of thing that can be in the field has a 'tag' and 'length'. The tag indicates which information comes next, and length indicates how many bytes that info occupies. The first number is always a tag. Here, '1' indicates a 'transaction public key'. Tx public keys are always 32 bytes, so we don't need to include the length. Thirty-two numbers later we find a new tag '2' which means 'extra nonce', its length is '9', and the next byte is '1' which means an 8-byte encrypted payment ID (the extra nonce can have fields inside it, for some reason). Eight bytes go by, and that's the end of this extra field. See [81] for more details. (note: in the original Cryptonote specification the first byte indicated the size of the field. Monero doesn't use that.) [140] src/cryptonote_basic/tx_extra.h
- `rct_signatures` (lines 40-50) - First part of signature data
- `type` (line 41) - Signature type; `RCTTypeBulletproof2` is type 4. Deprecated RingCT types `RCTTypeFull` and `RCTTypeSimple` were 1 and 2 respectively. Miner transactions use `RCTTypeNull`, type 0.
- `txnFee` (line 42) - Transaction fee in clear text, in this case 0.00003246 XMR
- `ecdhInfo` (lines 43-47) - 'elliptic curve diffie-hellman info': Obscured amount for each of the outputs $t \in \{0, ..., p-1\}$; here $p = 2$
- `amount` (line 44) - Field *amount* at $t = 0$ as described in Section 5.3
- `outPk` (lines 48-49) - Commitments for each output, Section 5.4
- `rctsig_prunable` (lines 51-132) - Second part of signature data
- `nbp` (line 52) - Number of Bulletproof range proofs in this transaction
- `bp` (lines 53-80) - Bulletproof proof elements (Bulletproofs were not explored in this document, so we will not itemize further)

$$\Pi_{BP} = (A, S, T_1, T_2, \tau_x, \mu, \mathbb{L}, \mathbb{R}, a, b, t)$$

- `MGs` (lines 81-129) - MLSAG signatures
- `ss` (lines 82-103) - Components $r_{i,1}$ and $r_{i,2}$ from the first input's MLSAG signature

$$\sigma_j(\mathfrak{m}) = (c_1, r_{1,1}, r_{1,2}, ..., r_{v+1,1}, r_{v+1,2})$$

- `cc` (line 104) - Component c_1 from aforementioned MLSAG signature
- `pseudoOuts` (lines 130-131) - Pseudo output commitments $C_j'^a$, as described in Section 5.3. Please recall that the sum of these commitments will equal the sum of the two output commitments of this transaction (plus the transaction fee commitment fH).

APPENDIX B

Block Content

In this appendix we show the structure of a sample block, namely the 1582196$^{\text{th}}$ 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).

```
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 the protocol version used to verify this block.
- `minor_version` (line 13) - Originally intended as a voting mechanism among miners, it now merely reiterates the `major_version`. Since the field does not occupy much space, the developers probably thought deprecating it would not be worth the effort.
- `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 17-40) - 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 the 1582256^{th} block, after 59 more blocks have been mined (it is a 60 block lock time since it can't be spent until 60 block time intervals have passed, e.g. $2 * 60 = 120$ minutes).
- `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 for which this miner tx's block reward was generated. 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 address assigning ownership of the miner tx's amount.
- `extra` (lines 33-36) - Extra information for the miner tx, including the transaction public key.
- `type` (line 38) - Type of transaction, in this case '0' for `RCTTypeNull`, indicating a miner tx.
- `tx_hashes` (lines 41-46) - All transaction IDs included in this block (but not the miner tx ID, which is `06fb3e1cf889bb972774a8535208d98db164394ef2b14ecfe74814170557e6e9`).

APPENDIX C

Genesis Block

In this appendix 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 [127]). 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 [13], and may operate a shady network of cryptocurrency-related software and services [134].

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 [127].

```
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,
```

```
14    "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 B, the structure appears basically the same. We point out some unique differences.

- `difficulty` (line 9) - The genesis block's difficulty is reported as 1, which means any nonce could work.
- `timestamp` (line 14) - The genesis block doesn't have a meaningful timestamp.
- `prev_id` (line 15) - We use an empty 32 bytes 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 were dispersed to Monero's founder thankful_for_today.
- `extra` (lines 33-36) - Using the encoding discussed in Appendix A, the genesis block's miner tx `extra` field just contains one transaction public key.
- `signatures` (line 37) - 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 39.