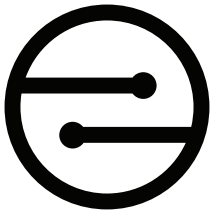

Mechanics of MobileCoin: First Edition

exploring the foundations of a private digital currency

June 12, 2023, v1.0.0

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

[1] ukoe@protonmail.com

[2] Author 'koe' worked on this document as part of a private contract with, then as an employee of, MobileCoin, Inc.

[3] kurt@oktav.se

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' containing public, distributed ledgers with records of currency transactions that can be verified by third parties [130].

It might seem at first glance that transactions need to be sent and stored in plain text format to make them publicly verifiable. In truth, 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 [170]. This is exemplified in the cryptocurrency MobileCoin.

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

For our experienced readers: MobileCoin is a standard one-dimensional directed acyclic graph (DAG) cryptocurrency blockchain [130], where blocks are consensuated with an implementation of the Stellar Consensus Protocol [115], transactions are validated in SGX secure enclaves [39] and are based on elliptic curve cryptography using the Ristretto abstraction [47] on curve Ed25519 [27], transaction inputs are shown to exist in the blockchain with Merkle proofs of membership [118] and are signed with Schnorr-style multilayered linkable spontaneous anonymous group signatures (MLSAG) [136], and output amounts (communicated to recipients via ECDH [57]) are concealed with Pedersen commitments [113] and proven in a legitimate range with Bulletproofs [35].

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 that 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 these goals without requiring the collaboration of any third party like a central authority, the currency's 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 typically 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

[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. A commodity or service whose individual units are so similar that one unit of the same grade or quality is considered interchangeable with any other unit of the same grade or quality. Examples: tin, grain, coal, sugar, money, etc." [14] 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' [124], 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 [143].

literally *publicizes* the complete transaction histories of its users.

To address the lack of privacy, users of cryptocurrencies such as Bitcoin can obfuscate transactions by using temporary intermediate addresses [132]. However, with appropriate tools, it is possible to analyze flows and to a large extent link true senders with receivers [155, 32, 139, 36].

In contrast, the cryptocurrency MobileCoin attempts to tackle the issue of privacy by storing only single-use addresses for ownership of funds in the blockchain, authenticating the dispersal of funds in each transaction with ring signatures, and verifying transactions within black-box 'secure enclaves' that discard extraneous information after verification. With these methods, there are no known effective ways to link receivers or trace the origin of funds.[3]

Additionally, transaction amounts in the MobileCoin blockchain are concealed behind cryptographic constructions, rendering currency flows opaque even in the case of secure enclave failures.

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

1.1 Objectives

MobileCoin is a new cryptocurrency employing a novel combination of techniques. Many of those techniques are either backed by technical documents missing key details pertinent to MobileCoin, or by non-peer-reviewed papers that are incomplete or contain errors.[4] Other aspects can only be understood by examining the source code and source code documentation (comments and READMEs) directly.

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

We intend to address this situation by introducing the fundamental concepts necessary to understand elliptic curve cryptography, reviewing algorithms and cryptographic schemes, and collecting in-depth information about MobileCoin'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 MobileCoin cryptocurrency.

In the first edition of this report, we have centered our attention on the launch version of the MobileCoin protocol[5,6], corresponding to version 1.0.1 of the MobileCoin software suite. All

[3] Depending on the behavior of users, if an attacker manages to break into secure enclaves there may be cases where transactions can be analyzed to some extent. For an example, see this article: [65].

[4] Seguias has created the excellent Monero Building Blocks series [154], which contains a thorough treatment of the cryptographic security proofs used to justify Monero's signature schemes. Seguias's series is focused on v7 of the Monero protocol, however much of what he discusses is applicable to MobileCoin, which uses many of the same building blocks as Monero.

[5] 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 1, which we call `TXTYPE_RCT_1` for clarity), which are general rules pertaining to how a transaction is constructed.

[6] In the launch protocol, blocks store a 'block version' set to 0. Semantically, this version number can be thought of as the number of protocol updates (i.e. scheduled hard forks) that have occurred since launch.

transaction and blockchain-related mechanisms described here belong to those versions.[7]

1.2 Readership

We anticipate many readers will encounter this report with little to no understanding of discrete mathematics, algebraic structures, cryptography[8], or blockchains. We have tried to be thorough enough that laypeople with a diversity of backgrounds may learn about MobileCoin 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.[9]

1.3 Origins of the MobileCoin cryptocurrency

MobileCoin's whitepaper was released in November 2017 by Joshua Goldbard and Moxie Marlinspike. According to the paper, their motivation for the project was to "develop a fast, private, and easy-to-use cryptocurrency that can be deployed in resource constrained environments to users who aren't equipped to reliably maintain secret keys over a long period of time, all without giving up control of funds to a payment processing service." [79]

1.4 Outline

Since MobileCoin is a very new cryptocurrency, we will not go into detail on hypothetical second-layer extensions and applications of the core protocol in this edition. Suffice it to say for now that topics like multisignatures, transaction proofs, and escrowed marketplaces are just as feasible for MobileCoin as they are for any well-designed progeny of the CryptoNote protocol.[10]

[7] The MobileCoin codebase's integrity and reliability relies on enough people reviewing it to catch most or all significant errors. We hope readers will not take our explanations for granted, and will instead verify for themselves the code does what it's supposed to. If it does not, we hope you will make a responsible disclosure (by emailing `security@mobilecoin.foundation`) for major problems, or Github 'issue' or 'pull request' (`https://github.com/mobilecoinfoundation/mobilecoin`) for minor issues.

[8] An extensive textbook on applied cryptography can be found here: [33].

[9] Some footnotes spoil future chapters or sections. These are intended to make more sense on a second read-through, since they usually tie local concepts to a broader context.

[10] Monero, initially known as BitMonero, was created in April 2014 as a derivative of the proof-of-concept currency CryptoNote [162]. Subsequent changes to Monero's transaction type retained parts of the original CryptoNote design (specifically, one-time addresses, ring signatures of one form or another, and key images). This means CryptoNote is in some sense an ancestor of MobileCoin, whose transaction scheme is inspired by Monero's `RCTTypeBulletproof2` transaction type.

1.4.1 Essentials

In our quest for comprehensiveness, we have chosen to present all the basic elements of cryptography needed to understand the complexities of MobileCoin, and their mathematical antecedents. In Chapter 2 we develop essential aspects of elliptic curve cryptography.

Chapter 3 expands on the Schnorr signature scheme introduced in the prior chapter, and outlines the ring signature algorithms that will be applied to achieve confidential transactions. Chapter 4 explores how MobileCoin 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. Chapter 6 is dedicated to membership proofs, which are used to prove MobileCoin transactions spend funds that exist in the blockchain. With all the components in place, we explain the transaction scheme used by MobileCoin in Chapter 7.

We shed light on secure enclaves in Chapter 8, the MobileCoin blockchain is unfolded in Chapter 9, and the MobileCoin consensus protocol used to create the blockchain is elucidated in Chapter 10.

1.4.2 Extensions

A cryptocurrency is more than just its protocol. As part of MobileCoin's original design, a 'service-layer' technology known as Fog was developed. Discussed in Chapter 11, Fog is a service that searches the blockchain and identifies transaction outputs owned by its users. This allows users to avoid scanning the blockchain themselves, which is time-consuming and resource-intensive (a prohibitive burden for e.g. mobile devices). Importantly, the service operator is not able to learn more than approximately how many outputs its users own.

1.4.3 Additional content

Appendix A explains the structure of a sample transaction that was submitted to the blockchain. Appendix B explains the structure of blocks in MobileCoin's blockchain. Finally, Appendix C brings our report to a close by explaining the structure of MobileCoin's origin (a.k.a. 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 MobileCoin implementation details can be found in the source code.[11] There is usually a file path, such as transaction/std/src/transaction_builder.rs, and a function, such as `create_output()`. Note: '-' indicates split text, such as Ristretto- Point → RistrettoPoint, and we neglect namespace qualifiers (e.g. `TransactionBuilder::`) in most cases.

Isn't this useful?

[11] Our margin notes are accurate for version 1.0.1 of the MobileCoin 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/mobilecoinfoundation/mobilecoin), so a complete history of changes is available.

Some code references are to third-party libraries, which we tagged with square brackets. These include

- [dalek25519]: the `dalek-cryptography curve25519-dalek` library [43]
- [dalekEd]: the `dalek-cryptography ed25519-dalek` library [44]
- [dalekBP]: the `dalek-cryptography bulletproofs` library [42]
- [blake2]: the Rust implementation [56] of hashing algorithm Blake2 [20]

To shorten their length, margin notes related to the transaction protocol use the tag [MC-tx], which stands for the directory path 'transaction/core/'. Code from MobileCoin's Fog implementation [69] is tagged with [MC-fog]. Finally, functions that are executed exclusively within SGX secure enclaves are marked with an asterisk '*'.

1.5 Disclaimer

All signature schemes, applications of elliptic curves, and 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 implementation details can be found in the source code. In particular, as a common saying among cryptographers and security engineers 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.[12] Moreover, original contributions in this document may not be well-reviewed and are likely untested, so readers should exercise their judgement when encountering them.

1.6 History of 'Mechanics of MobileCoin'

'Mechanics of MobileCoin: First Edition' is an adaptation of 'Zero to Monero: Second Edition', published in April 2020 [100]. 'Zero to Monero' itself (its first edition was published in June 2018 [19]) is an expansion of Kurt Alonso's master's thesis, 'Monero - Privacy in the Blockchain' [18], published in May 2018.

There are several notable differences between 'Mechanics of MobileCoin: First Edition' and 'Zero to Monero: Second Edition'.

- Parts of the core content were improved or adjusted for MobileCoin. For example, there is an updated group theory section, and also a discussion of the Ristretto abstraction, in Chapter 2. More generally, the early chapters have been edited in various small ways.

[12] Cryptographic primitives are the building blocks of cryptographic algorithms. For example, in elliptic curve cryptography the primitives include point addition and scalar multiplication on that curve (see Chapter 2).

- Transaction details specific to MobileCoin replaced details specific to Monero (for example, the addition of membership proofs in Chapter 6, which serve a purpose similar to Monero's output offsets).
- MobileCoin's consensus protocol (an implementation of the Stellar Consensus Protocol [115]) replaced Monero's more standard mining-based (Nakamoto [130]) consensus mechanism (Chapter 10).
- MobileCoin uses secure enclaves extensively (Chapter 8). They have an important role in MobileCoin's privacy model and are essential to the Fog technology (Chapter 11).

1.7 Acknowledgements

This report, like 'Zero to Monero' before it, would not exist without Alonso's original master's thesis [18], so to him, I (koe) owe a great debt of gratitude. Robb Walters got me involved with MobileCoin, and is without doubt a force for good in this crazy world. All I can feel is admiration for the team at MobileCoin, who have designed and built something that goes far beyond what anyone could reasonably hope for in a cryptocurrency. Finally, it is hard to express in words how incredible the legacy of modern technology is. MobileCoin would truly be impossible without the prior research and work of countless people, only a tiny subset of whom can be found in this document's bibliography.

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 MobileCoin 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 that allow a point to be represented 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 MobileCoin, it will typically be BLAKE2b[2], but if not explicitly mentioned then it is not important to the theory.

[blake2] src/blake2b.rs

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 (called preimage resistance), have an interesting feature known as the *large avalanche effect* that 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/n. Let's imagine a number line. To calculate $c = a \pmod{n}$, we stand at point a, then walk

[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}. [98]

[2] The BLAKE2 hashing algorithm is a successor to the NIST standard *SHA-3* [15] finalist BLAKE. The BLAKE2b variant is optimized for 64-bit platforms. [22]

toward zero with each step $= n$ until we reach an integer ≥ 0 and $< n$. That is c. For example, 4 (modulo 3) $= 1$, $-5 \pmod 4 = 3$, and so on.

Formally, the positive modulus is here defined for $c = a \pmod n$ as $a = nx + c$, where $0 \leq c < n$ and x is a signed integer that gets discarded (n 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. Essentially, we are converting $7 * 8 \pmod 9$ into

$$\begin{aligned} &1 * 2^0 * 8 + 1 * 2^1 * 8 + 1 * 2^2 * 8 + 0 * 2^3 * 8 \\ &= 8 + 16 + 32 + 0 * 64 \end{aligned}$$

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: $i = (0, ..., A_{size} - 1)$

[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.

   (a) If A[i] $\stackrel{?}{=} 1$, then $r = r + s \pmod{n}$.
   (b) Compute $s = s + s \pmod{n}$.

2. Use the final r: $c = r$.

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] $\stackrel{?}{=} 1$, 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 [156].[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* [5]:

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

More generally (and more rapidly), the so-called 'extended Euclidean algorithm' [4] 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 (which, as we will see, is not a standalone operation defined for finite fields).

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

The same principle would apply to something like $x = (a - b * c * d)^{-1}(e * f + g^h) \pmod n$.[6]

[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 modulus of large numbers can exploit modular equations. For example, we can write $254 \pmod{13} \equiv 2 * 10 * 10 + 5 * 10 + 4 \equiv (((2) * 10 + 5) * 10 + 4) \pmod{13}$. 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 $(\text{mod } q)$.

[dalek25519] src/backend/serial/[u32|u64]/field.rs

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

Typically, an elliptic curve is defined as the set of all points with coordinates (x, y) satisfying a *Weierstraß* equation [83] (for a given (a, b) pair):[7]

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

The cryptocurrency MobileCoin uses a special curve belonging to the category of so-called *twisted Edwards* curves [26], 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 [28] 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[8]

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

These formulas for addition also apply to point doubling; that is, when $P_1 = P_2$. To subtract a point, invert its coordinates over the y-axis, $(x, y) \to (-x, y)$ [26], and use normal point addition. Recall that 'negative' elements $-x$ of $\mathbb{F}_q$ are really $-x \ (\text{mod } q)$.

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.

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

[8] Typically elliptic curve points are converted into projective coordinates (or a similar representation, e.g. extended twisted Edwards coordinates [84]) prior to curve operations like point addition, in order to avoid performing field inversions for efficiency. [172]

2.3.2 Group theory

Importantly, elliptic curves have what is known as an *abelian group* structure [1]. Every curve has a so-called 'point-at-infinity' I, which is like a 'zero position' on the curve (its coordinates are $(0, 1)$), and a finite number of points N that can be computed. Any point P added with itself N times will produce the point-at-infinity I, and $I + P = P$.

Group theory: Intro

For now, let's step back from elliptic curves and imagine a clock-like ring with *order* N.[9] The zeroth position 0 (or I) is at the top, followed by 1 and preceded by $N - 1$. Clearly, we can walk around the ring with step-size 1, and reach the top again after N steps.

While it's useful to think of ourselves as walking around the circle, it's much more accurate to think about each position on the circle as being a 'point', and taking steps is like 'adding points together'. If we stand at point P_2 on a circle with $N = 6$, then with each 'step' we are adding P_2 to the point we are currently standing on.

$$I + P_2 = P_2$$
$$P_2 + P_2 = P_4$$
$$P_4 + P_2 = P_6 = I$$

Note how we landed back on our starting position after three steps. The point P_2 has generated a *cyclic* subgroup with order 3 out of multiples of itself. It's cyclic because after a while you always get back to where you started. It's a subgroup since it doesn't (necessarily) contain all the points on the circle.

The order of any point is equal to the number of points in the subgroup it can generate. If a point's order is prime, then all the other (non-point-at-infinity) points it generates will generate the same subgroup. In our previous example, where the subgroup's order was 3, the point P_4 also generates the same subgroup.

$$I + P_4 = P_4$$
$$P_4 + P_4 = P_2$$
$$P_2 + P_4 = P_6 = I$$

However, P_1 has order 6 which is not prime, so not all of the points it generates have order 6 (only P_5 will generate the same subgroup).

Group theory: Useful concepts

We always land somewhere on the ring no matter how many multiples of a point we add together. This lets us simplify scalar multiplication from nP to $[n \pmod u]P$, where u is the order of the

[9] The basics of group theory are very important to grasp for the rest of this document. Visual learners may find it helpful to draw pictures and work out what is happening by hand.

point P. A point can't actually be multiplied by 0, so if $n \pmod u \stackrel{?}{=} 0$ just multiply P by u or return the 0^{th} position directly (I). The orders of all possible subgroups are divisors of N (by *Lagrange's theorem* [110]).

To find the order, u, of any given point P's subgroup:

1. Find N (e.g. use *Schoof's algorithm* [160]).
2. Find all the divisors of N.
3. For every divisor n of N, compute nP.
4. The smallest n such that $nP \stackrel{?}{=} I$ is the order u of the subgroup.

Suppose we are given two points P_a, P_b and are told they both have order u. Do they necessarily belong to the same subgroup, or might there be more than one subgroup with order u?

We can think of the two points in terms of P_1 which has order N, such that $P_a = n_a * P_1$ and $P_b = n_b * P_1$. We know that $N * P_1 = I$, and that $u * P_a = u * P_b = I$. Therefore $u * n_a$ and $u * n_b$ must be multiples of N.

Since u is a divisor of N (recalling Lagrange's theorem), for $u * n_a$ and $u * n_b$ to be multiples of N, scalars n_a, n_b must have a common denominator that is another divisor of N, namely $e = N/u$. Therefore $P_a = (n_a/e) * e * P_1$ and $P_b = (n_b/e) * e * P_1$, or in other words P_a and P_b are multiples of the same point $e * P_1$ and must both be members of that point's subgroup.

Put simply, any two points P_a and P_b with order u are in the same subgroup, which is composed of multiples of $(N/u) * P_1$. Furthermore, for any random point $P' = n' * P_1$, the expression $(N/u) * P'$ will either be a point in the u subgroup (since $n' * (N/u) * P_1$ is a multiple of $(N/u) * P_1$), or I (in which case n' must be a multiple of u, so P' is a member of the $e = (N/u)$ subgroup).

Group theory: Back to elliptic curves

Elliptic curve points have no concept of 'proximity', so for our clock-like example with $N = 6$, P_3 is no 'closer' or 'farther' from I than P_1. However, to connect the analogy we can 'map' curve points onto the ring. Take any point P_w with order N and put it at position 1, then construct the ring out of multiples of P_w. All of the observations we have made so far still hold, and will hold even if the mapping is redone with a different point P_z that also has order N.

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.[10] 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 $P = nG$.

[dalek25519] src/backend/serial/curve_models/mod.rs

Based on our understanding from the previous section, we can use the following algorithm to find (non-point-at-infinity) points in the subgroup of order l:

[10] EC with small cofactors allow relatively faster point addition, etc. [26].

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 $P \stackrel{?}{=} I$, return to step 2; otherwise, 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).[11] Scalar multiplication can be seen as a *one-way function*, which paves the way for using elliptic curves for cryptography.[12]

[dalek25519] src/backend/serial/[u32|u64]/scalar.rs

The scalar product nP is equivalent to $(((P + P) + (P + P))...)$. 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 $n_{scalar} \rightarrow n_{binary}$; $A = [n_{binary}]$; $R = I$, the point-at-infinity; $S = P$
2. Iterate through: $i = (0, ..., A_{size} - 1)$
   (a) If A[i] $\stackrel{?}{=} 1$, 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 operations between scalars are mod l.

2.3.3 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*.[13] 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.

[11] In modular arithmetic, finding the discrete log of h with respect to g, x, such that $g^x = h$, is thought to be difficult for some group orders. [58]

[12] 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.

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

2.3.4 Diffie-Hellman key exchange with elliptic curves

A basic *Diffie-Hellman* [57] 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.[14]

2.3.5 Schnorr signatures and the Fiat-Shamir transform

In 1989, Claus-Peter Schnorr published a now-famous interactive authentication protocol [150], generalized by Maurer in 2009 [112], that allows someone to prove they know the private key k of a given public key K without revealing any information about it [114]. It goes something like this:

1. The prover generates a random integer $\alpha \in_R \mathbb{Z}_l$,[15] computes αG, and sends αG to the verifier.

2. The verifier generates a random *challenge* $c \in_R \mathbb{Z}_l$ and sends c to the prover.

3. The prover computes 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).

[14] The DHP is thought to be of at least similar difficulty to the DLP, although it has not been proven. [76]

[15] 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. [dalek25519] src/scalar.rs `Scalar::random()`

If α was chosen randomly by the prover, then r is randomly distributed [151] and k is information-theoretically secure within r (it can still be found by solving the DLP for K or αG).[16] 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).[17]

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

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*. [114]

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 [114].[18]

Using a hash function, instead of the verifier, to generate challenges is known as a *Fiat-Shamir transform* [63], because it makes an interactive proof non-interactive and publicly verifiable [114].[19,20]

Non-interactive proof

1. Generate random number $\alpha \in_R \mathbb{Z}_l$, and compute αG.
2. Calculate the challenge using a cryptographically secure hash function, $c = \mathcal{H}(T_p, [\alpha G])$.[21]
3. Define the response $r = \alpha + c * k$.
4. Publish the proof pair $(\alpha G, r)$.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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 more later (Section 3.4).

[21] MobileCoin has a policy of 'domain separating' [164] different uses of hash functions. This in practice means prefixing each 'use case' of a hash function with a unique bit-string. We model it here with the tag T_p, which might be the text string "simple Schnorr proof". Domain separated hash functions have different outputs even with the same inputs. For the remainder of this document, we leave out domain separation tags for succinctness. In most cases, uses of hash function can be assumed to have their own tag, but readers seeking precision should refer to the source code. [MC-tx] src/domain_separators.rs

Verification

1. Calculate the challenge: $c' = \mathcal{H}(T_p, [\alpha G])$.
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.

[dalek25519] src/edwards.rs

2.3.6 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 [92], ANSI X9.62, and [83] for more on this topic.

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), the signing method, $\mathfrak{m}$, 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 to know 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 and Ristretto

MobileCoin uses a particular twisted Edwards elliptic curve for cryptographic operations, *Ed25519*, the *birational equivalent*[22] of the Montgomery curve *Curve25519*. It actually uses Ed25519 indirectly via the Ristretto encoding abstraction, which we will discuss. Both Curve25519 and Ed25519 were released by Bernstein *et al.* [26, 27, 28].

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

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:

[dalek25519] src/constants.rs `#[test] test_d_vs_ratio()`

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

This curve addresses many concerns raised by the cryptography community.[23] It is well known that NIST[24] 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 [81]. 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 [10].

Curve Ed25519 is not subject to any patents (see [99] for a discussion on this subject), and the team behind it has developed and adapted basic cryptographic algorithms with efficiency in mind [28].

[dalek25519] src/edwards.rs

Twisted Edwards curves have order expressible as $N = 2^c l$, where l is a prime number and c is a positive integer. In the case of curve Ed25519, its order is a 76-digit number (l is 253 bits):[25]

$$2^3 \cdot 7237005577332262213973186563042994240857116359379907606001950938285454250989$$

[dalek25519] src/constants.rs `BASEPOINT_ORDER`

2.4.1 Problem with cofactors

As mentioned in Section 2.3.2, only points in the large prime-order subgroup of a given elliptic curve are used in cryptographic algorithms. It is therefore sometimes important to make sure a given curve point belongs to that subgroup [82].

For example, it is possible to have multiple curve points that, when multiplied by the same scalar n, create the same resultant point. This can occur when n is a multiple of h and the points are created as sums between a prime-order point and different cofactor-order points. Multiplying by n causes the cofactor-order component to 'disappear'.[26]

To be clear, given some point K in the subgroup of order l, any point K^h with order h, and an integer n divisible by h:

$$\begin{aligned} n * (K + K^h) &= nK + nK^h \\ &= nK + 0 \end{aligned}$$

[23] Even if a curve appears to have no cryptographic security problems, it's possible the person/organization that created it knows a secret issue which 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. [146]

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

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

[26] Cryptocurrencies that inherited the CryptoNote code base had an infamous vulnerability related to adding cofactor-subgroup points to normal-subgroup points. It was solved by checking that key images (discussed in Chapter 3) are in the correct subgroup with the test $l\tilde{K} \stackrel{?}{=} 0$ [67].

The importance of using prime-order-subgroup points was the motivation behind Ristretto, which is an encoding abstraction that can be applied to some elliptic curves (e.g. Ed25519). Ristretto efficiently constructs a prime-order group out of an underlying non-prime group. [50]

2.4.2 Ristretto

Ristretto can be thought of as a 'binning' procedure for points in the underlying curve. The number of bins is equal to the prime-order subgroup of the relevant curve (l), and each bin has h elements.[27] Curve operations 'on' or 'between' bins behave just like curve operations on the prime-order subgroup, except in this model rather than a specific point, the result is a specific bin.

This implies the members of a bin must be variants of a prime-order subgroup point (in other words, a prime-order point plus all the members of the cofactor-order subgroup, including the point-at-infinity). Given two bins, adding any of their members together will land you in the same third bin. In this way, curve operations on 'Ristretto points', which are simple containers that can hold a bin member from any bin, behave just like operations on prime-order subgroup points.

For example, given Ristretto points

$$P_1 = P_1^{prime} + P_1^{cofactor}$$
$$P_2 = P_2^{prime} + P_2^{cofactor}$$

their sum $P_1 + P_2 = P_3$ will be

$$P_3^{prime} = P_1^{prime} + P_2^{prime}$$
$$P_3^{cofactor} = P_1^{cofactor} + P_2^{cofactor}$$

Here P_3^{prime} defines which bin you landed in, and $P_3^{cofactor}$ corresponds to the specific member that was created.

Two members of the same bin are considered 'equal'. There is a relatively cheap way to test equality, which we describe in Section 2.4.4.

In each bin the point from the prime-order subgroup is the bin's 'representative canonical member'. Moreover, that point can be easily found by 'compressing' any of the bin members and then 'decompressing' the result. This way curve points can be communicated in compressed form, and recipients can be assured they are handling effectively prime-order points and don't have to be concerned about cofactor-related problems.

As a bit of callback, given some point K in the prime-order subgroup and two points K_a^h, K_b^h in the cofactor-order subgroup, both $K + K_a^h$ and $K + K_b^h$ will compress and then decompress into the same point K.

Since all bin members get compressed to the same bit string, there is no 'gotcha' (as there is with standard Ed25519) where presenting different compressed members of a bin to a byte-aware context (e.g. a hash function or byte-wise comparison) will have different results.

[27] In group theory, what we call bins are more correctly known as 'cosets' [173].

2.4.3 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 the Ristretto *point compression* technique, described below, however, it is possible to reduce this amount by half, to 32 bytes.

2.4.4 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 [47], but we can give a brief insight into how it works. Normal point compression for the Ed25519 curve was standardized in [96], first described in [27], and the concept was introduced in [123].

As background, it's helpful to know that the normal point compression scheme follows from a transformation of the twisted Edwards curve equation (wherein $a = -1$): $x^2 = (y^2-1)/(dy^2+1)$,[28] which indicates there are two possible x values ($+$ or $-$) for each y. Field elements x and y are calculated (mod q), so there are no actual negative values. However, taking (mod q) of $-x$ will change the value between odd and even since q is odd. For example: 3 (mod 5) = 3, -3 (mod 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.

Ristretto has a different approach to compressing points, where the sign of the x coordinate is not encoded.

Assume we want to compress a point (x, y). First we transform it into *extended twisted Edwards coordinates* [84] $(X : Y : Z : T)$, where $XY = ZT$ and $aX^2 + Y^2 = Z^2 + dT^2$.

$$\begin{aligned} X &= x \\ Y &= y \\ Z &= 1 \\ T &= xy \pmod{q} \end{aligned}$$

Square Root: `Sqrt(u, v)`

1. Create an algorithm for computing $\sqrt{u/v} \pmod{q}$.[29]

[dalek25519] src/field.rs `FieldElement::sqrt_ratio_i()`

[28] Here $d = -\frac{121665}{121666}$.

[29] These algorithms are merely shown for a sense of completeness. It's best to consult the `dalek` library's implementation [43], the Ristretto Group's notes [47], and the Ristretto/Decaf IETF draft [52] for any production-level applications.

2. Define $i = 2^{(q-1)/4} \pmod q$.
3. Compute[30] $z = uv^3(uv^7)^{(q-5)/8} \pmod q$. Set `nonzero-square = true`.
   (a) If $vz^2 \stackrel{?}{=} u \pmod q$, set $r = z$.
   (b) If $vz^2 \stackrel{?}{=} -u \pmod q$, calculate $r = i * z \pmod q$.
   (c) If $vz^2 \stackrel{?}{=} i * (-u) \pmod q$, calculate $r = i * z \pmod q$ and set `nonzero-square = false`.
4. If the least significant bit of r is 1 (i.e. it is odd), return $-r$, otherwise return r. Also return the boolean `nonzero-square`.[31]

Encoding

[dalek25519] src/ristretto.rs `RistrettoPoint::compress()`

1. Define
   (a) $u_1 = (Z + Y) * (Z - Y) \pmod q$
   (b) $u_2 = XY \pmod q$
2. Let inv = $\texttt{Sqrt}(1, u_1 u_2^2) \pmod q$.
3. Define
   (a) $i_1 = u_1 * \text{inv} \pmod q$
   (b) $i_2 = u_2 * \text{inv} \pmod q$
   (c) $z_{inv} = i_1 i_2 T \pmod q$
4. Let b equal the least significant bit of $z_{inv} * T \pmod q$
   (a) If $b \stackrel{?}{=} 1$, define
      i. $X' = Y * 2^{(q-1)/4} \pmod q$
      ii. $Y' = X * 2^{(q-1)/4} \pmod q$
      iii. $D' = i_1 * \texttt{Sqrt}(1, a - d) \pmod q$
   (b) Otherwise if $b \stackrel{?}{=} 0$, define
      i. $X' = X$
      ii. $Y' = Y$
      iii. $D' = i_2$
5. If $z_{inv} * X' \pmod q$ is odd, set $Y' = -Y' \pmod q$.
6. Compute $s = D' * (Z - Y') \pmod q$. If s is odd, set $s = -s \pmod q$.
7. Return s.

Decoding

[30] Since $q = 2^{255} - 19 \equiv 5 \pmod 8$, $(q-5)/8$ and $(q-1)/4$ are straightforward integers.

[31] According to the comments in `[dalek] src/field.rs FieldElement::sqrt_ratio_i()`, only the 'positive' square root should be returned, which is defined by convention in [27] and [52] as field elements with the least significant bit not set (i.e. 'even' field elements). In normal Ed25519 point decompression [27] we would compute $\texttt{Sqrt}(y^2 - 1, dy^2 + 1)$, then decide whether to use the 'positive'/'negative' result variant depending on if we want the even/odd x coordinate. Basically, a compressed point is the y coordinate, with the most significant bit equal to 0 or 1 to indicate if the point's x coordinate is even/odd.

1. Given a supposed compressed curve point s. Reject s if it is odd or $\geq q$.
2. Compute
$$y = \frac{1 + as^2}{1 - as^2} \pmod q$$
3. Compute (x should be 'even' after this step) the following. If `nonzero-square` is `false`, then the point is invalid.
$$x = \texttt{Sqrt}(4s^2, ad(1 + as^2)^2 - (1 - as^2)^2) \pmod q$$
4. Convert to extended coordinates. If T is odd, then the point is invalid.
$$X = x$$
$$Y = y$$
$$Z = 1$$
$$T = xy \pmod q$$

[dalek25519] src/ristretto.rs `CompressedRistretto::decompress()`

We can use extended coordinates to test if two points belong to the same Ristretto bin. If either $X_1Y_2 \stackrel{?}{=} Y_1X_2$ or $Y_1Y_2 \stackrel{?}{=} -aX_1X_2$ holds, then the points $P_1 = (X_1 : Y_1 : Z_1 : T_1)$ and $P_2 = (X_2 : Y_2 : Z_2 : T_2)$ are 'equal' for our purposes [48].[32,33]

[dalek25519] src/ristretto.rs `RistrettoPoint::ct_eq()`

Implementations of Ed25519 typically use the generator $G = (x, 4/5)$ [27], where x is the 'even' variant based on normal point decompression (footnote 31 from earlier in this section describes how it works) of $y = 4/5 \pmod q$. The Ristretto generator is straightforwardly the bin that contains G, and the point selected to represent it is G itself.

[dalek25519] src/constants.rs `RISTRETTO_BASEPOINT_POINT`

2.5 Binary operator XOR

The binary operator XOR is a useful tool that will appear in Section 5.3. It takes two arguments and returns true if one, but not both, of them is true [12]. 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:
$$\begin{aligned}\text{XOR}(\{1,1\},\{1,0\}) &= \{1+1, 1+0\} \pmod 2 \\ &= \{0,1\}\end{aligned}$$

[32] Since $a = -1$, the second test simplifies to $Y_1Y_2 \stackrel{?}{=} X_1X_2$.

[33] Multiple extended coordinates can represent a given curve point [49], so X may not always equal x, and the same for Y and y. This equality test works for all extended coordinate representations.

Each of these also produce $\{0,1\}$: XOR($\{1,0\},\{1,1\}$), XOR($\{0,0\},\{0,1\}$), and XOR($\{0,1\},\{0,0\}$). For XOR inputs with b bits, there are 2^{b} total combinations of inputs that would make the same output. This means if $C = \mathrm{XOR}(A,B)$ and input $A \in_R \{0,...,2^{\mathrm{b}}-1\}$, an observer who learns C would gain no information about B (its real value could be any of 2^b possibilities).

At the same time, anyone who knows two of the elements in $\{A,B,C\}$, where $C = \mathrm{XOR}(A,B)$, can calculate the third element, such as $A = \mathrm{XOR}(B,C)$. XOR indicates if two elements are different or the same, so knowing C and B is enough to expose A. A careful examination of the truth table reveals this vital feature.[34]

[34] One interesting application of XOR (unrelated to MobileCoin) 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. However, 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 MobileCoin'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 2.3.4), 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 2.3.5) across all bases.[1] Assume the existence of a hash function $\mathcal{H}_n$ mapping to integers from 0 to $l - 1$.[2]

[1] We could turn this proof into a signature by including a message `m` in the challenge hash. The terminology proof/signature is loosely interchangeable in this context.

[2] MobileCoin uses the `dalek` library's hash-to-scalar function $\mathcal{H}_n(x)$ = `from_hash<Blake2b>`(x). The input x is hashed by the BLAKE2b hashing algorithm [22] into a 64-byte bit string, which is passed to `from_bytes_mod_order_wide()` where it is reduced modulo l. The final result is in the range 0 to $l - 1$ (although it should really be 1 to $l - 1$). [dalek25519] src/scalar.rs `from_hash<D>()`

1. Generate random number $\alpha \in_R \mathbb{Z}_l$, then 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 2.3.5). 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 use multiple private keys but only return one response, you would need to compute an appropriate α_j for each one based on the challenge, 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 [37]), 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 [107] building on the work of Rivest *et al.* in [145]. The authors detailed a group signature algorithm called LSAG characterized by three properties: *anonymity, linkability,* and *spontaneity.*[4] 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 MobileCoin, they will ultimately allow for unforgeable, signer-ambiguous transactions that leave currency flows largely untraceable even in the case of secure enclave failures.

[4] A spontaneous signature is one that can be constructed without the cooperation of any non-signer (e.g. any third party). [107]

Ring signatures primer

The feasibility of ring signatures follows from one simple observation. As long as the challenge used to define a Schnorr signature's response both depends on αG and is uniformly distributed, we have a lot of freedom when defining it. A simple example would be hashing the challenge a second time (compared to normal), as in the following made-up signature scheme.

1. Generate random number $\alpha \in_R \mathbb{Z}_l$, and compute αG.
2. Calculate the intermediate challenge, $c_i = \mathcal{H}(\mathfrak{m}, [\alpha G])$.
3. Calculate the real challenge, $c_r = \mathcal{H}([c_i G])$.
4. Define the response, $r = \alpha - c_r * k$.
5. Publish the signature (c_r, r).

A verifier would then compute:

1. Calculate the intermediate challenge: $c_i' = \mathcal{H}(\mathfrak{m}, [rG + c_r * K])$.
2. Calculate the real challenge: $c_r' = \mathcal{H}([c_i' G])$.
3. If $c_r = c_r'$, then the signature passes.

If, however, the real challenge computation is indistinguishable from the intermediate challenge computation, then it becomes possible to hide which one corresponds to the true signer (given a real signer K_r and fake signer K_f). Here is an example signature scheme.

1. Generate random number $\alpha \in_R \mathbb{Z}_l$, and compute αG.
2. Calculate the intermediate challenge, $c_i = \mathcal{H}(\mathfrak{m}, [\alpha G])$.
3. Generate an intermediate response $r_i \in_R \mathbb{Z}_l$. Calculate the real challenge based on fake signer K_f,
$$c_r = \mathcal{H}(\mathfrak{m}, [r_i G + c_i * K_f])$$
4. Define the real response, $r_r = \alpha - c_r * k_r$.
5. Randomize the order of tuples $\{c_i, r_i, K_f\}$, $\{c_r, r_r, K_r\}$ and assign them numbers 1 or 2. Publish the signature $(c_1, r_1, r_2, K_1, K_2)$.

The verifier computes:

1. Calculate the second challenge: $c_2' = \mathcal{H}(\mathfrak{m}, [r_1 G + c_1 * K_1])$.
2. Calculate the first challenge: $c_1' = \mathcal{H}(\mathfrak{m}, [r_2 G + c_2' * K_2])$.
3. If $c_1 = c_1'$, then the signature passes, and the verifier can't tell if K_1 or K_2 was the signer.

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 who verified it would not be able to tell which ring member was the actual signer.

Our Schnorr-like signature scheme in Section 2.3.6 can be considered a one-key ring signature. As discussed in the primer, 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 verify it with $\mathcal{R}$ 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. Looking back to 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 remaining ring signature schemes discussed in this chapter display several properties that will be useful for producing confidential transactions.[5] Note that both 'signer ambiguity' and 'unforgeability' also apply to SAG signatures.

[5] 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 MobileCoin'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.[6] MobileCoin 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.[7] As we will show, this property is used to prevent double-spending attacks in MobileCoin (except with negligible probability).

Unforgeability No attacker can forge a signature except with negligible probability.[8] This is used to prevent theft of MobileCoin funds by those not in possession of the appropriate private keys.

In the LSAG signature scheme [107], the owner of a private key could produce one anonymous unlinked signature per ring.[9] In this section we present an enhanced version of the LSAG algorithm where linkability is independent of the ring's decoy members.

The modification was formalized in [136] based on a publication by Adam Back [23] regarding the CryptoNote [170] ring signature algorithm (previously used in various CryptoNote derivatives, and now mostly/entirely deprecated), which was in turn inspired by Fujisaki and Suzuki's work in [75].

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 $\mathcal{H}_p$ that maps to curve points in EC.[10,11]

[6] 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 MobileCoin it is the ring size. Expanding anonymity sets makes it progressively harder to track down real actors. As we will see in Chapter 7, MobileCoin users potentially belong to much larger anonymity sets, even on the order of 'all MobileCoin users'.

[7] 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.

[8] Certain ring signature schemes, including the one in MobileCoin, 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 [136] and [107].

[9] 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.

[10] It doesn't matter if points from $\mathcal{H}_p$ are compressed or not. They can always be decompressed. In our case, the function outputs a point in extended coordinates.

[11] MobileCoin 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 Ristretto-flavored hash-to-point algorithm Elligator in [46], which is used by MobileCoin, and note that the input is hashed to 64 bytes with BLAKE2b before being passed to Elligator. [dalek25519] src/ristretto.rs `from_hash<Blake2b>()`

1. Calculate key image $\tilde{K} = k_\pi \mathcal{H}_p(K_\pi)$.[12]
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:[13]

1. For $i = 1, 2, ..., n$ iteratively compute, replacing $n + 1 \to 1$,
$$c'_{i+1} = \mathcal{H}_n(\mathfrak{m}, [r_i G + c_i K_i], [r_i \mathcal{H}_p(K_i) + c_i \tilde{K}])$$
2. 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 could 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 [97] suggests it may not be necessary, although adding the prefix is standard practice for similar signature schemes (LSAG uses key prefixing).

[12] In MobileCoin it is important to use the hash-to-point function for key images instead of a static base point so linearity doesn't lead to linking signatures created by the same address (even if for different one-time addresses). See [170] page 18.

[13] As discussed in Section 2.4.1, with normal Ed25519 adding cofactor-order points to a big-prime-order point can create multiple 'equivalent' points when multiplied by a multiple of h. It is important here since if we add a cofactor-order point to $\tilde{K}$ and all c_i are multiples of h (which we could achieve with automated trial and error using different α and r_i values), we could make h unlinked valid signatures using the same ring and signing key [67]. However, using the Ristretto abstraction to manage EC keys solves this problem by forcing signature creators to communicate key images in compressed form, which always decompress to the same bin member no matter what cofactor-order point was added pre-compression (recall Section 2.4.2).

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).[14]

3.5 Multilayer Linkable Spontaneous Anonymous Group (MLSAG) signatures

In order to sign transactions, one has to sign with multiple private keys. In [136], 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 $i = \pi$. MLSAG has a generalized 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[15]

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

[14] LSAG, which is quite similar to bLSAG, is unforgeable, meaning no attacker could make a valid ring signature without knowing a private key. Liu *et al.* prove forgeries that manage to pass verification are extremely improbable [107].

[15] In MobileCoin MLSAGs, key prefixing is done by baking the ring members directly into the message added to each challenge. We go into more detail in Section 7.2.2. One idiosyncrasy to keep in mind, however, is that the key image is explicitly prefixed in the challenge hash, and isn't included in the message. As we will explain in Section 7.2.2, MobileCoin only needs one key image per signature.

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

[MC-tx] src/ring_signature/mlsag.rs `RingMLSAG::sign()`

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

Why it works

Just as with the 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.[16]

Space requirements

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

[16] 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.

CHAPTER 4

Addresses

The ownership of digital currency stored in a blockchain is controlled by so-called 'addresses'. Addresses are sent money that only the address-holders 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 MOB".

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 MOB".

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 7 (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 9.

[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, MobileCoin users have two sets of private/public keys, (k^v, K^v) and (k^s, K^s), generated as described in Section 2.3.3.

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).[3,4]

account-keys/src/account_keys.rs *struct* PublicAddress

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).[5]

4.2 One-time addresses

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

The address is never used directly.[6] Instead, a Diffie-Hellman-like exchange is applied between the address and a so-called txout public key. The result is a unique *one-time address* for each transaction output to be paid to the user. This means even external observers who know all users' addresses cannot use them to identify which user owns any given transaction output.[7]

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

[3] To communicate an address to other users, it is common to encode it in base58, a binary-to-text encoding scheme first created for Bitcoin [16]. In brief, addresses are set in a 'protobuf' format [55], which is just a method of writing messages so they are easy to understand by recipients, a 4-byte checksum is prepended to the protubuf'd address (allowing users of the address to check if the message they receive is malformed), then the resulting byte sequence is converted to base58.

mobilecoind/src/service.rs `get_public_address_impl()`

[4] Alongside the view and spend keys, standard MobileCoin addresses include information related to the Fog service (see Section 11.1).

[5] In the core MobileCoin implementation, the view and spend private keys for a given 'account' are derived from a BIP-39-compliant 'mnemonic' [30], which is a set of 24 words selected randomly from a pre-defined list. The mnemonic (converted from a word-list to a 'seed' [111] as defined by BIP-39) is the first element of a BIP-32 'path' [29] with components 'mnemonic-seed/44/866/account_index', where '44' signifies this path is BIP-44-compliant [31], '866' is the MobileCoin 'coin-type' registered with SLIP-0044 [147], and 'account_index' is the account number associated with this mnemonic (you can derive multiple accounts from the same mnemonic). The path is passed to a SLIP-0010-compliant [148] key-derivation function (that employs HMAC-SHA512 [103]) [61], which basically hashes the path in a sequence of hash-steps and spits out a `Slip10Key`. This `Slip10Key` is hashed with HKDF-SHA512 [64, 104], using two different domain-separation strings ("mobilecoin-ristretto255-view" and "mobilecoin-ristretto255-spend") to create the account's private view and spend keys. A person only needs to save their mnemonic and account index in order to access (view and spend) all of the outputs they own (spent and unspent) [101].

account-keys/slip10/src/lib.rs `Slip10Key::into()` `derive_slip10_key()`

[6] 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.

[7] Except with negligible probability.

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 [170]:

1. Alice generates a random number $r \in_R \mathbb{Z}_l$, and calculates the one-time address[8]
$$K^o = \mathcal{H}_n(rK_B^v)G + K_B^s$$
[MC-tx] src/onetime_keys.rs `create_onetime_public_key()`

2. Alice sets K^o as the addressee of the payment, adds the output amount '0' and the so-called *transaction output public key* rG (henceforth abbreviated as the txout public key) to the transaction data, and submits it to the network.[9]

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 $K_B'^s \stackrel{?}{=} K_B^s$, he knows the output is addressed to him.

   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.

   [MC-tx] src/onetime_keys.rs `view_key_matches_output()`

4. The one-time keys for the output are:
$$K^o = \mathcal{H}_n(rK_B^v)G + K_B^s = (\mathcal{H}_n(rK_B^v) + k_B^s)G$$
$$k^o = \mathcal{H}_n(rK_B^v) + k_B^s$$
[MC-tx] src/onetime_keys.rs `recover_onetime_private_key()`

To spend his '0' amount output 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.

As will become clear in Chapter 7, 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.[10,11]

[8] In MobileCoin whenever an EC key is hashed, it is the Ristretto compressed form of that key (unless otherwise stated). Aside from being a simple convention, this ensures there are no cofactor-order-point related problems.

[9] In CryptoNote the key rG is termed the 'transaction public key' [170], while MobileCoin re-brands it to 'transaction output public key' to better express its role.

[10] 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. MobileCoin avoids this problem by enforcing unique txout public keys on the blockchain. Even if duplicate one-time keys exist, only one of them can be paired with the txout public key that helped create it, so Bob won't ever find the other one and needn't worry about it.

*consensus/service/src/validators.rs `is_valid()`

[11] 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 partial read access to the user's transaction history, without any further rights. However, the private view key can't be used to recreate key images (see Section 7.2.3), so it can't reveal when outputs have been spent. This third party would also be able to decrypt the output's amount (see Section 5.3).

4.3 Subaddresses

MobileCoin users can generate subaddresses from each address [135]. 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 that a given subaddress is derived from any particular address [135].[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.

Bob generates his i^{th} subaddress ($i = 1, 2, ...$) from his address as a pair of public keys $(K^{v,i}, K^{s,i})$: account-keys/src/account_keys.rs `subaddress()`

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

So,

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

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
$$K^o = \mathcal{H}_n(rK_B^{v,1})G + K_B^{s,1}$$

2. Alice sets K^o as the addressee of the payment, adds the output amount '0' and the txout public key $rK_B^{s,1}$ to the transaction data, and submits it to the network. [MC-tx] src/onetime_keys.rs `recover_public_subaddress_spend_key()`

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 $K_B'^s = K^o - \mathcal{H}_n(rK_B^{v,1})G$. When he sees that $K_B'^s \stackrel{?}{=} K_B^{s,1}$, he knows the transaction is addressed to him.[13] mobilecoind/src/database.rs `get_subaddress_id_by_spk()`

[12] The alternative to subaddresses would be generating many normal addresses. To view each address's balance, you would need to do a separate scan of the blockchain record (very inefficient). With subaddresses, users can maintain a look-up table of spend keys, so one scan of the blockchain takes the same amount of time for 1 subaddress, or 10,000 subaddresses.

[13] An advanced attacker may be able to link subaddresses [59] (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

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

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

$$K^o = \mathcal{H}_n(rK_B^{v,1})G + K_B^{s,1} = (\mathcal{H}_n(rK_B^{v,1}) + k_B^{s,1})G$$
$$k^o = \mathcal{H}_n(rK_B^{v,1}) + k_B^{s,1}$$

4.4 Multi-output transactions

mobilecoind/src/payments.rs `build_tx_proposal()`

Most transactions will contain more than one output, if nothing else, to transfer 'change' back to the sender (see Section 5.4).[14]

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, MobileCoin uses a unique txout private key r_t for each output $t \in 1, ..., p$. The key r_tG is published as part of its corresponding output in the blockchain.

[MC-tx] std/src/transaction_builder.rs `create_output()`

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

Moreover, to promote uniformity the core MobileCoin implementation never passes users' normal addresses out for receiving funds. Instead, only subaddresses are made available.[15,16] Since cross-wallet compatibility is a guiding principle in wallet design, it is unlikely that any new wallet will break that pattern. This means, unlike other CryptoNote variants [170], the txout public key will likely never appear in the form r_tG in the blockchain, but will instead always be $r_tK_t^{s,i}$.

4.5 Multisignature addresses

Sometimes it is useful to share ownership of funds between different people/addresses. MobileCoin does not currently support any form of multisignatures, although they could theoretically be

with $K^o = \mathcal{H}_n(rK_B^{v,2})G + K_B^{s,1}$ and includes txout 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 view 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). Based on an extensive analysis [166], the most efficient known mitigation is encrypting the txout private key r and adding it to transaction data. See [134] for background on this topic. We are not aware of any wallets that have implemented a mitigation for this attack. See Chapter 11 footnote 16 for a different mitigation that could be added to MobileCoin in the future.

[14] Each transaction is limited to no more than 16 outputs.

[MC-tx] src/constants.rs `MAX_OUTPUTS`

[15] The txout public key is constructed differently for subaddresses vs. normal addresses, so if normal address are permitted then to properly construct transactions senders must know what kind of address they are dealing with.

[16] At the time of MobileCoin's launch, existing wallets used subaddress 0 (i.e. index 0) as the 'normal/main' subaddress, subaddress 1 as the 'change' subaddress (for receiving all change outputs), and subaddress 2 as the second normal subaddress (see `DEFAULT_CHANGE_SUBADDRESS_INDEX` in [137] for example). This may seem wonky, but is now a de facto standard, so all future wallet implementations should use the same scheme. Note that MobileCoin wallets only have one 'account' per view/spend private key pair. This is unlike Monero wallets, where a given pair's subaddress space is divided into 'accounts', and outputs in each 'account' are segregated from each other. As a consequence, MobileCoin wallets only need one change subaddress per key pair, whereas Monero wallets have a change subaddress for each account associated with a key pair.

implemented by a third party. Future editions of 'Mechanics of MobileCoin' may discuss the topic, but for now we defer to the treatment in Chapters 9 and 10 of [100].

CHAPTER 5

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 MobileCoin 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' that 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 alongside 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 Alice has a secret string "My secret" and the value she wants to commit to is *heads*. She hashes $h = \mathcal{H}(\text{"My secret"}, \mathit{heads})$ and gives h to Bob. Bob flips a coin, then Alice tells Bob the secret string "My secret" and that she committed to *heads*. Bob calculates $h' = \mathcal{H}(\text{"My secret"}, \mathit{heads})$. If $h' = h$, then he knows Alice called *heads* before the coin flip.

Alice uses the so-called 'salt', "My secret", 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* [140] 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 [113, 151].

5.3 Amount commitments

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

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

[dalekBP] src/generators.rs `PedersenGens::commit()`

[MC-tx] src/ring_signature/mod.rs `GENERATORS`

[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, $x \to xG$. In other words, after transforming a scalar into a curve point, addition operations between points proceed in 'parallel' to scalar additions. The transformation of scalar $a + b$ always equals $aG + bG$ based on the individual transformations of scalars a and b.

[3] In the case of MobileCoin, $H = H_p(G)$.

[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' [171]. Even the committer can't find another combination without solving the DLP for γ, a property called 'computational binding' [171].

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 $r_t K_B^{v,t}$ using the 'transaction output public key' (recall Section 4.4). Every output in the blockchain has a mask y_t that senders and receivers can privately compute, and a $masked\_value_t$ stored in the transaction's data. While y_t is an elliptic curve scalar and occupies 32 bytes, b_t will be restricted to 8 bytes by the range proof so only an 8-byte value needs to be stored.[5,6]

$$y_t = \mathcal{H}_n(\text{“}mc\_amount\_blinding\text{”}, [r_t K_B^{v,i}])$$
$$masked\_value_t = b_t \oplus_8 \mathcal{H}_n(\text{“}mc\_amount\_value\text{”}, [r_t K_B^{v,i}])$$

[MC-tx] src/amount/mod.rs `Amount::new()`

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 $masked\_value_t$ to reveal b_t.

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

[MC-tx] src/amount/mod.rs `Amount::get_value()`

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 copies of other transactions' outputs (telling validators 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), a txout public key (for accessing the output via Diffie-Hellman exchange), an output commitment hiding the amount, the encoded output amount from Section 5.3, and a so-called 'encrypted fog hint' (an 84-byte field we discuss in Section 11.2.3).

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. MobileCoin uses a technique called RingCT [136] to accomplish this.

If we have a transaction with m inputs containing amounts $a_1, ..., a_m$, and p outputs with amounts $b_1, ..., b_p$, then an observer would justifiably expect that:[7]

[5] Domain separation tags are written explicitly here to emphasize that the hash results are not the same.

[6] We include the index t (from within the transaction that created a given output) for clarity, but this information is lost when an output is published in a block, since outputs are sorted by txout public key within each block to ensure all validator nodes create the same block.

*consensus/enclave/impl/src/lib.rs `form_block()`

[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.

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

Since commitments are additive and we don't know γ, we could easily prove that 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:

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

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 7 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 the outputs in a transaction to the blockchain, the network will want to verify that 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:

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

Fortunately, choosing such blinding factors is easy. In the current version of Mobilecoin, all blinding factors are random except for the m^{th} pseudo out commitment, where x_m' is simply

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

[MC-tx] src/ring_signature/rct_bullet-proofs.rs `verify()` `sign_with_balance_check()`

[8] Recall from Section 2.3.1 that 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 $(\text{mod } q)$, so $(-x \pmod q)$.

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 MOB (over 7.2x10^{74}!) than we put in.

The solution addressing this issue in MobileCoin 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 [35] (and also explored in [171, 41]).[9] Given the involved and intricate nature of Bulletproofs, the scheme is not elucidated in this document. Moreover we feel the cited materials adequately present its concepts.

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[10,11]

$$\Pi_{BP} = (A, S, T_1, T_2, t_x, t_x^{blinding}, e^{blinding}, \mathbb{L}, \mathbb{R}, a, b)$$

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).[12] 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 $2^{64} - 1$.

The n-tuple Π_{BP} occupies $(2 \cdot \lceil \log_2(64 \cdot (m + p)) \rceil + 9) \cdot 32$ bytes of storage.[13]

[dalekBP] src/range_proof/mod.rs `prove_multiple_with_rng()`

[MC-tx] src/range_proofs/mod.rs `check_range_proofs()`

[MC-tx] src/ring_signature/rct_bulletproofs.rs `verify()`

[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 amount 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 Ed25519, l is about 2^189 times bigger than the available range, which means it would take a ridiculous 2^189 outputs to counterfeit money.

[10] 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.

[11] The variables in a Bulletproof are unrelated to other variables in this document. Symbol overlap is merely coincidental. Note that in MobileCoin transactions, Bulletproofs are represented by a byte blob that gets passed to the `dalek-cryptography bulletproofs` library [42] for verification.

[12] 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 there is no theoretical limit to how many can be batched together. Currently in MobileCoin, each transaction is only allowed one Bulletproof, which is validated by itself without batching. As we will discuss more in the coming chapters, MobileCoin nodes discard Bulletproofs after validating them, so batching would only be beneficial if the volume of transactions is significant.

[13] In the initial version of MobileCoin, pseudo output commitments are also range proofed in the same way as output commitments, which is why there is an 'm' in $(64 \cdot (m + p))$. Apparently the MobileCoin development team plans to stop range proofing pseudo output commitments in the next version of the protocol [120], which seems to us like a good idea since range proofing them serves no purpose.

CHAPTER 6

Membership Proofs

A transaction input is just an old output, so verifiers have to check that transaction inputs actually exist in the blockchain. In MobileCoin, transactions are validated inside so-called *secure enclaves* (see Chapter 8), which are intended to be opaque boxes not open to observers. Doing so makes it possible to hide which inputs were present in a transaction from everyone except the transaction author herself.[1]

Enclaves are far too small to contain copies of the entire blockchain, which must be stored in more 'visible' parts of a node's machine. If an enclave has to reach out to the local chain to check if a transaction's inputs are legitimate, then it would be easy for the machine's owner to figure out what they are.

We resolve this dilemma by, instead of referencing where old outputs can be found, making copies of them and constructing *membership proofs* that show they belong to the blockchain. A membership proof is just a Merkle proof [118].

6.1 Merkle trees

Merkle trees [118] are a simple and effective way of taking a large data set and representing it with a much smaller identifier [165]. They are well-known and see widespread use among blockchain-based technologies [142].

[1] As will be discussed more over the next few chapters, after a transaction has been validated most of its content gets discarded. A transaction that goes into an enclave leaves truncated.

6.1.1 Simple Merkle trees

A Merkle tree is a binary hash tree. With a data set containing (for example) items {A, B, C, D, E, F, G}, the tree is constructed by hashing each pair consecutively. When no more hashes are possible, you have reached the *root hash* which represents the entire data set. In the following diagram a black arrow indicates a hash of inputs.

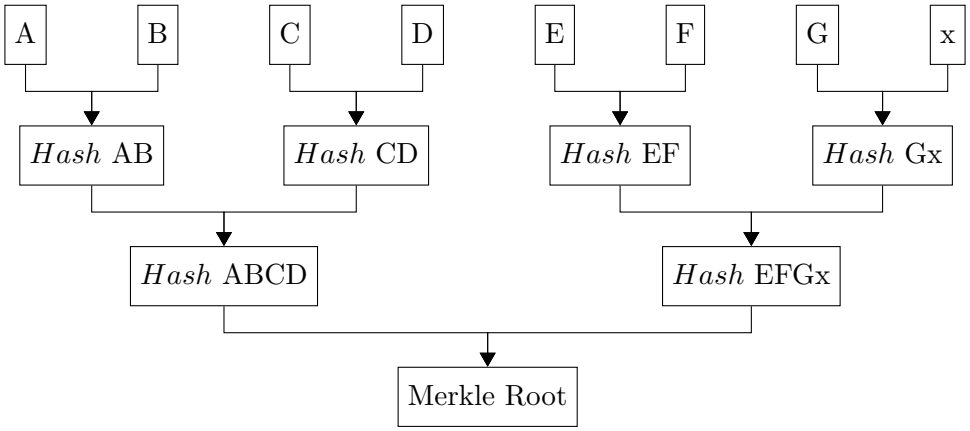

Diagram 6-1: Merkle Tree

We append a 'filler', or 'nil', value called x to the end so the tree is a complete binary structure. Fillers aren't a mandatory part of Merkle trees, but will be useful in MobileCoin membership proofs.

Usually items are hashed individually before being hashed into a pair [142, 165]. It isn't that important for our discussion here whether items {A, B, C, ...} represent data objects or their hashes.[2] The original inputs to a Merkle tree are called 'leaf nodes', middle hashes are 'intermediate nodes', filler nodes that don't represent any data elements (at any level of the tree) are 'nil nodes', and the final hash is a 'root node'.[3]

6.1.2 Simple inclusion proofs

To prove that element C belongs to the data set, only nodes {C, D, *Hash* AB, *Hash* EFGx} are required. A verifier may reconstruct the prover's Merkle root from those nodes, and compare it with the expected value.

[2] In the case of MobileCoin, leaf nodes are always hashes of the data objects they represent.

[3] For MobileCoin Merkle proofs, the hashes to create leaf nodes, intermediate nodes, and nil nodes are each domain separated (the root hash is treated as an intermediate node). Doing so provides 'defense-in-depth' [68, 85] against second preimage attacks [66] alongside the de facto defense of implementation robustness.

[MC-tx] src/membership_proofs/mod.rs

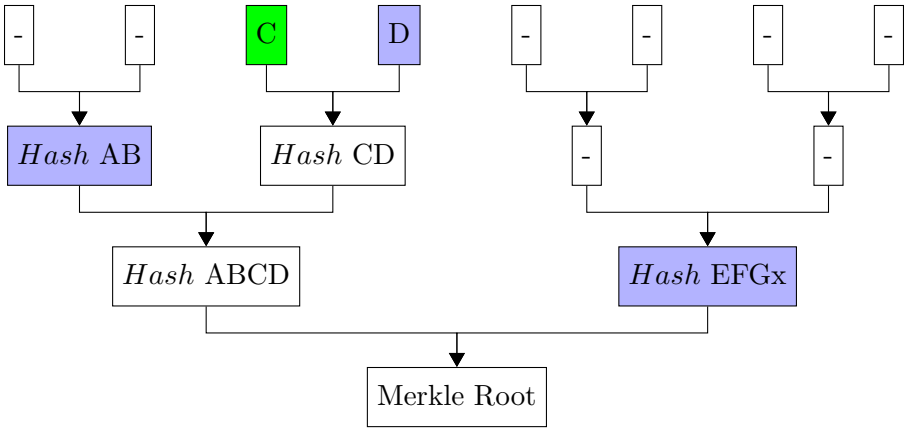

Diagram 6-2: Proof of Membership

In our example, the input elements are three layers deep. A Merkle proof for any node m layers deep requires $m+1$ nodes (including itself) to verify. In other words, if there are $2^{n-1} < N <= 2^n$ total items in the data set, it will take $n+1$ nodes to construct a given leaf node's proof. Since we use fillers, the leaf node layer will always have 2^n entries.

6.2 Membership proofs

To prove elements B and G belong to the same data set, we only need two Merkle proofs that create the same root hash. This offers a method for secure enclaves to verify that a given output belongs to the blockchain without leaking to its local machine which output it is verifying.

In broad strokes, the enclave receives a transaction containing an output with a Merkle proof of membership. It requests a Merkle proof from the local machine corresponding to some other output. Finally, it compares the resulting root hashes of the two proofs. If they match, then the transaction's output must belong to the local machine's data set.

[MC-tx] src/validation.rs `validate_membership_proofs()`

6.2.1 Highest index element proofs

When new elements are appended to a data set, root hashes for new Merkle proofs will not match root hashes for older proofs. This means, given an old proof, we can't verify it by simply obtaining a new proof for any random element. Instead we must get a proof *as if created at the same time as our old proof.*

One approach is to pretend all the new elements added since our old proof was made don't exist, and use that truncated data set to construct a proof for a random element. However, if the data set contains 4.3 billion elements (2^{32}), then a single proof from scratch will cost almost 8.6 billion hashes (one per leaf node, plus all the intermediate nodes)!

Instead, we precompute all the nodes in the data set's Merkle tree, continuously updating the nodes that change whenever a new element is added. To validate an old membership proof, we get a new proof for the element that was at the very end of the data set (e.g. blockchain) when our to-be-validated proof was made.

ledger/db/src/tx_out_store.rs `push()`

Within any data set, the last element is unique. Its proof does not contain any 'partial' nil nodes. All proof nodes are either 'complete' (representing a full set of elements), or 'empty' (representing filler elements).[4] Take the following tree for example (Diagram 6-3), with highest element E's proof nodes highlighted.

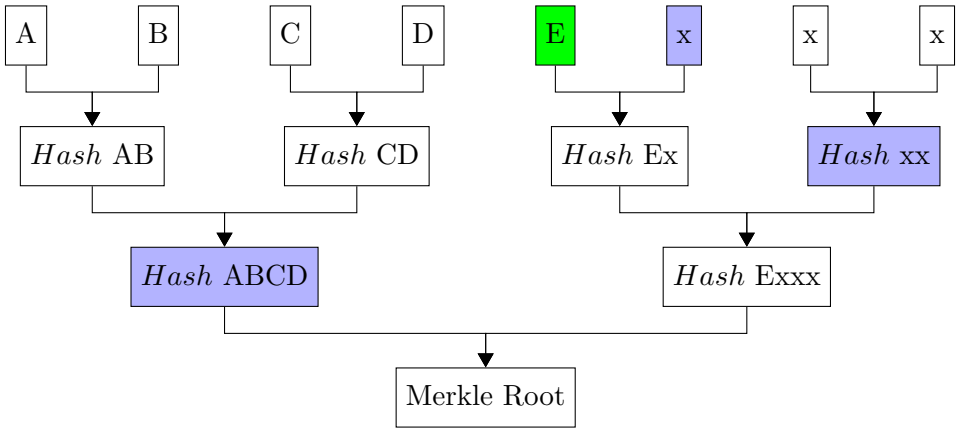

Diagram 6-3: Highest Element Membership Proof

The absence of partial nil nodes is significant because it lets us easily 'simulate' any state of the data set. Using a normal, up-to-date proof for the element that was at the highest index of that old data set, treat all 'right-hand' nodes (i.e. nodes 'above' the proof element) in the proof as nil nodes. The root hash of that simulated proof equals the old data set's root hash.

[MC-tx] src/membership_proofs/mod.rs `derive_proof_at_index()`

Current proofs that are convertible to simulated proofs have an additional property. Suppose you have multiple old proofs each made at a different state of the data set. The current proofs used to verify them will be on different elements (corresponding to different old highest indices). However, the non-simulated versions of those proofs all have the same root hash (representing the current data set). This is useful in MobileCoin because current proofs are obtained from outside secure enclaves. By comparing the proofs' root hashes, validator enclaves can be sure all old proofs are verified with respect to the same data set.[5]

*consensus/enclave/impl/src/lib.rs `tx_is_well_formed()`

One final note. The size of the binary tree for a proof is based on the size of its data set. In other words, how many entries the leaf node layer has is the next power of two above (or equal to, if

[4] In MobileCoin, all pure nil nodes are identical, no matter what level of the tree they are found in. Partial nil nodes are constructed like normal as a hash of inputs (e.g. a hash of normal node and pure nil node).

[5] Comparing current proof root hashes occurs in two places in MobileCoin. When a transaction is validated, current proofs obtained by the validating enclave from the local machine must have the same root hash. Then when a block is being assembled, the current proofs stored with all transactions' inputs in that block must have the same root hash. These requirements make it more difficult for validator node operators to trick their enclaves or observers of the blockchain (see Section 9.6).

[MC-tx] src/membership_proofs/mod.rs `hash_nil()`
*consensus/enclave/impl/src/lib.rs `form_block()`

your indexing starts at 1) the highest element's index. This is true for simulated proofs as well, so the number of elements in the 'current' data set is irrelevant for verifying old proofs.

6.2.2 MobileCoin membership proofs

The MobileCoin blockchain consists mainly of transaction outputs, which are added sequentially as new transactions are submitted and verified (see Chapter 9). However, as mentioned, transaction verifiers (secure enclaves, see Chapter 8) are unable to read the blockchain directly. As such, old outputs spent by transactions must have corresponding membership proofs that show they exist in the blockchain record.

Membership proofs include the highest output index at the time they were produced. To verify a membership proof, the verifying secure enclave obtains a current proof from the insecure local machine for the output at that old proof's 'highest index', simulates the old data set, and compares the old and simulated proofs' root hashes. Since the current proof reveals almost nothing about the membership proof being verified (only approximately how old it is), the outputs being spent in a transaction are safely hidden from node operators.[6]

consensus/ service/src/ tx_manager/ mod.rs `is_well_formed()`

A MobileCoin transaction input's membership proof has the following content.

[MC-tx] src/tx.rs

1. `index`: The on-chain index $i^{element}$ of the output being proven a member of the blockchain.
2. `highest_index`: The index $i^{highest}$ of the last output in the blockchain at the time this proof was created. It is used to make a comparison proof for validating this membership proof.

*consensus/ service/src/ validator.rs `well_formed_check()`

3. Proof elements: one per node in the proof.[7]
   (a) `range`: The range of elements this node represents. It is a pair of indices $[i^{low}, i^{high}]$. A node ranging [0, 3] represents elements {0, 1, 2, 3}. The range is used to determine if a node is on the 'left' or 'right' when creating its parent node. For example, with two nodes X and Y ranging [0, 3] and [4, 7], they are composed into XY = $\mathcal{H}$(X, Y), with the lower range on the left.[8,9]
   (b) `hash`: The node hash h^{node}, used in combination with other node hashes to reconstruct the proof's root hash.

[MC-tx] src/membership_proofs/ mod.rs `compose_adjacent_membership_elements()`

[6] Membership proofs for different old outputs will have different contents. If, given two membership proofs with the same number of nodes, the time it takes to compute their root hashes is different, observers may be able to use those timing differences as the basis for an analysis of their contents (a so-called 'side-channel' attack [133]). For that reason membership proof validation is designed to be 'constant-time' with respect to node values and ranges.

[7] In MobileCoin transactions, a copy of the input (a.k.a. the old output) is stored separately from its membership proof. Representing a mild case of duplication, the output's leaf node hash is included among the proof elements.

[8] The hash function for making nodes is $\mathcal{H}_{Blake2b256}()$, which is a wrapper around Blake2b that creates 32-byte outputs instead of 64-bytes. The hash wrapper is used by the RustCrypto library `KDFs` [64, 104] to hash the inputs. Leaf nodes are $\mathcal{H}_{Blake2b256}(\mathcal{H}_{32}(\texttt{TxOut}))$, using the hash function $\mathcal{H}_{32}()$ to condense transaction outputs. $\mathcal{H}_{32}()$ is a hash function that outputs a 32-byte digest. Internally it uses the *Keccak* hashing algorithm, albeit via the 'Merlin transcript' interface [51] provided by the `dalek-cryptography merlin` library [45].

[9] Technically element ranges could be inferred from each node's position in the element list, along with the highest index. Perhaps future versions of the MobileCoin protocol will simplify membership proofs by making ranges implicit.

CHAPTER 7

Ring Confidential Transactions (RingCT)

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

"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 has txout public key $r_t K^{s,i}$ and 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 the message signed 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.

7.1 Transaction types

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

In this report, we have focused our attention on *Ring Confidential Transactions* [136], a.k.a. *RingCT*, as they are implemented in the current version of MobileCoin. The flavor of RingCT we have discussed so far, and which will be further elaborated in this chapter, is based on Monero's transaction type `RCTTypeBulletproof2` [100]. We name it `TXTYPE_RCT_1` to represent that it's the first variant of RingCT implemented in MobileCoin.

transaction/std/src/transaction_builder.rs `build()`

We present a conceptual summary of transactions in Section 7.5.

7.2 Ring Confidential Transactions of type `TXTYPE_RCT_1`

Currently (protocol v1) all new transactions must use this transaction type, in which each input is signed separately. An actual example of a `TXTYPE_RCT_1` transaction, with all its components, can be inspected in Appendix A.

7.2.1 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 those 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 output amounts are *lower* in total than transaction inputs, in order to make it costly for attackers to flood the network and blockchain with transactions. Transaction fee amounts f are stored in clear text in the transaction data transmitted to the network. Block validators will create an additional output consuming the fee (see Section 9.4 for more on fees).

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

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 $b_1, ..., b_p$. Let these new commitments be $C^b_1, ..., C^b_p$.

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 that 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:

[MC-tx] src/ring_signature/rct_bullet-proofs.rs `verify()`

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

7.2.2 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.[1,2] To sign input j, she combines a set of size v with her own j^{th} unspent one-time address (placed at unique index π) into a *ring*, along with false commitments to zero, as follows:

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

Alice uses an MLSAG-inspired 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. Input j's signature's 'round hash' looks like this:[3]

[MC-tx] src/ring_signature/mlsag.rs `sign_with_balance_check()`

$$c_{i+1} = \mathcal{H}_n(\mathtt{m}, \tilde{K_j}, [r_{i,1}G + c_i K^o_{i,j}], [r_{i,1}\mathcal{H}_p(K^o_{i,j}) + c_i \tilde{K_j}], [r_{i,2}G + c_i(C_{i,j} - C'^a_{\pi,j})])$$

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.[4] Since

[1] In the initial version of MobileCoin, it is standard for the sets of 'additional unrelated addresses' to be selected at random from the set of outputs existing in the chain. This method permits the 'guess-newest' heuristic, which says the most 'recent' output in a ring is likely to be the true signer about 80% of the time [122]. Since MobileCoin transactions are validated in secure enclaves, an analyst of the MobileCoin blockchain can only use that heuristic if they break into a secure enclave participating in the MobileCoin network. Future versions of MobileCoin will hopefully improve the ring member selection algorithm, for example by selecting randomly from 'nearby' the real output being spent. By nearby we just mean outputs added to the blockchain around the same time, so they have similar indices. This method is essentially a binning procedure where the entire ring is a bin. First, select a random offset in the range $[-x, x]$ and add it to the real output's on-chain index. That offsetted index is the center of the bin, which ranges $[-x, x]$ around its center. Randomly select indices from within the bin for use as decoy ring members. Another option is selecting from a gamma distribution over the set of on-chain outputs [122] (used in Monero [100]).

mobilecoind/src/payments.rs `get_rings()`

[2] In Nakamoto-based cryptocurrencies, it is typically necessary to wait for several blocks after obtaining an output before it can be spent. For example, Monero enforces a default spendable age of 10 blocks at the protocol level [100] (equating to 20 minutes of delay). Since MobileCoin uses the Stellar Consensus Protocol, which has a different threat model than Nakamoto consensus, it is considered safe to spend outputs as soon as they are acquired (which in practice can result in delays on the order of 5-10 seconds between receipt and spend of an output).

[3] As mentioned in footnote 15 of Chapter 3, key images aren't part of the message signed, but instead each input's key image is added to its MLSAG's challenge hashes explicitly.

[4] 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 (i.e. one giant MLSAG-like signature containing

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 (and key images, since those are key prefixed explicitly).[5] 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_j is the core of MobileCoin's transaction model. Signing $\mathfrak{m}$ with k^o_j proves you are the owner of the amount committed to in C^a_j. Verifiers can be confident that transaction authors are spending their own funds without knowing which funds are being spent, how much is being spent, or what other funds they might own!

7.2.3 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 valid MLSAG signature's key images must have been deterministically produced from legitimate private keys.

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. If they have, then we can be sure we are witnessing an attempt to re-spend an output $(C^a_{\pi,j}, K^o_{\pi,j})$.

consensus/service/src/validators.rs `is_valid()`

7.3 Space requirements

MobileCoin does not use the common Nakamoto consensus mechanism for consensuating transactions, employed by cryptocurrencies like Bitcoin [130] and Monero [170]. In those blockchains it is important for the network to retain complete copies of all transactions so new participants can validate the chain independently. However, in MobileCoin once a transaction has been validated, most of its data is discarded, and never gets stored in the blockchain (see Chapters 9 and 10).

---

all rings in parallel). This means even if one input's origin becomes identifiable, the other inputs' origins will not be.

[5] The actual message is $\mathfrak{m} = \{\mathcal{H}_{32}(\texttt{TxPrefix}), \texttt{pseudo\_output\_commitments}, \texttt{range\_proof\_bytes}\}$ where:
$\mathcal{H}_{32}()$ is a transcript-based hash function mentioned in a footnote of Section 6.2.2
`TxPrefix` = {vector of `TxIn` for inputs, vector of `TxOut` for outputs, fee, tombstone block}
`TxIn` = {copies of each ring member's `TxOut`, `TxOutMembershipProof`s for each ring member}
`TxOut` = {`Amount` (amount commitment and encrypted output amount), `target_key` (one time address), `public_key` (txout pub key), `e_fog_hint`}
See Appendix A regarding this terminology.

[MC-tx] src/ring_signature/rct_bullet-proofs.rs `extend_message()`

While transaction data may be discarded after validation, exactly how large a transaction is has implications for the network burden of passing it around.

MLSAG signature

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

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

With this in mind and assuming point compression (Section 2.4.4), an input signature σ_j will require $(2(v+1)+1)*32$ bytes. On top of this, the key image $\tilde{K}^o_{\pi,j}$ and the pseudo output commitment $C'^a_{\pi,j}$ leave a total of $(2(v+1)+3)*32$ bytes per input.

Note that verifying all of a `TXTYPE_RCT_1` transaction's MLSAGs includes the computation of $(C_{i,j} - C'^a_{\pi,j})$ for each ring member, and the final balance check $(\sum_j C'^a_j \stackrel{?}{=} \sum_t C^b_t + fH)$.

Range proofs

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

[MC-tx] src/range_proofs/mod.rs `check_range_proofs()`

Outputs

An output consists of an output commitment and 8-byte encrypted amount, a one-time address and txout public key, and an 84-byte encrypted fog hint (see Section 11.2.3). This comes out to $p*(3*32+8+84) = p*188$ bytes for all the outputs in a transaction.

[MC-tx] src/tx.rs *struct* `TxOut`

Inputs

Transaction verifiers (secure enclaves) are not expected to have free access to the blockchain, so they must get copies of input ring members from the transaction author. A transaction input contains copies of the old outputs used as ring members (including the real output being spent), and membership proofs for each of them.

Recalling Section 6.2.2, a membership proof contains two 8-byte indices and a set of 'membership elements'. A membership element has a 'range' which is two 8-byte indices, and a 32-byte hash. Membership elements correspond to nodes in the Merkle proof, and if the Merkle tree is 31 nodes deep (which would be possible if the blockchain had on the order of 2 billion outputs, not an unreasonable number in the long run), then you would require 32 elements for the entire proof. Therefore, for a Merkle tree n nodes deep, one proof is $(16 + (n+1)*(16+32))$ bytes.

Each ring member has a membership proof and 188-byte output copy, so a transaction input is $(v+1)*(252+48*n)$ bytes.

On-chain storage

Only the information necessary to make new transactions is saved in the blockchain. Specifically, only key images and outputs are saved, so only $m * 32 + p * 188$ bytes per transaction are stored in the blockchain.[6]

ledger/db/src/lib.rs *struct* `LedgerDB`

7.4 Miscellaneous semantic rules in MobileCoin

Even with robust cryptographic mechanisms for preserving the privacy of users, seemingly harmless implementation details can give away a lot of information about what they are up to [144, 166].

To promote transaction uniformity regardless of who makes them, MobileCoin has a number of purely semantic rules governing how one may be constructed.[7]

[MC-tx] src/validation/validate.rs

1. A transaction is limited to a maximum of 16 inputs and 16 outputs.
2. Rings for input MLSAGs must have exactly 11 members.
3. Each ring member must be unique in a transaction (no duplicates within or between rings).
4. Ring members within a ring are sorted based on their txout public keys (a ring member is just an old output from the chain). This sorting also applies to the real signer.
5. Inputs are sorted based on the txout public key of their first ring member.
6. Key images and the outputs' txout public keys must be unique within a transaction.[8]

[6] With a ring size of 11, Merkle tree depth of 31, and 16 inputs and outputs, one full transaction will be 323,040 bytes. After discarding non-essential information, it reduces to 3,520 bytes.

[7] In the launch version of MobileCoin, transaction outputs are not required to be sorted (although they are sorted by default when constructing a transaction). Hopefully future versions of the protocol will add that requirement (see [119]).

transaction/std/src/transaction_builder.rs `build()`

[8] Checking for key image and txout public key uniqueness within a transaction may appear superfluous, as duplicates aren't permitted at the blockchain level. However, since a transaction is added to the chain 'all at once', checking for duplicates between the chain and transaction won't catch duplicates within the transaction itself. A similar check is performed at the block level (no duplicates within the block) for the same reason.

consensus/enclave/impl/src/lib.rs `form_block()`

7.5 Concept summary: MobileCoin transactions

To close out this chapter, we present the main content of a transaction, organized for conceptual clarity. A real example can be found in Appendix A.

- Type: `TXTYPE_RCT_1`
- Inputs [`TxIn`]: for each input $j \in \{1, ..., m\}$ spent by the transaction author
  - **Ring member output copies**: an output $\texttt{TxOut}_i$ for $i \in \{1, ..., v+1\}$
  - **Ring member membership proofs**: proving the output copies can be found in the blockchain, there is a proof for each $i \in \{1, ..., v+1\}$
    - *index*: $i_i^{element}$
    - *highest index*: $i_i^{highest}$
    - *elements*: for a Merkle tree n nodes deep, $n+1$ proof elements $(i^{low}, i^{high}, h^{node})$
- Outputs [`TxOut`]: for each output $t \in \{1, ..., p\}$ to subaddress $(K_t^{v,i}, K_t^{s,i})$
  - **One-time address**: $K_t^{o,b}$ for output t
  - **Txout public key**: $r_t K_t^{s,i}$ for output t
  - **Output commitment**: C_t^b for output t
  - **Encoded amount**: so output owners can compute b_t for output t
    - *Masked value*: $b_t \oplus_8 \mathcal{H}_n(\text{“}mc\_amount\_value\text{”}, [r_t K_t^{v,i}])$
  - **Encrypted fog hint**: $\texttt{e\_fog\_hint}_t$ to help third-parties collect and store output t[9]
- Transaction proofs [`SignatureRctBulletproofs`]
  - **Inputs are owned and unspent**: for each input $j \in \{1, ..., m\}$
    - *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
  - **Output amounts are in a legitimate range**: an aggregate Bulletproof for all output commitments with amounts b_t and pseudo output commitments with amounts a'_j
    - *Range proof*: $\Pi_{BP} = (A, S, T_1, T_2, t_x, t_x^{blinding}, e^{blinding}, \mathbb{L}, \mathbb{R}, a, b)$
- Transaction fee: A value communicated in clear text multiplied by 10^{12} (i.e. in ‘picoMOB’ atomic units, see Section 9.3.1), so a fee of 1.0 would be recorded as 1000000000000.
- Tombstone block: A future block’s index selected by the transaction author. Each transaction must be added to the blockchain before its tombstone block is created, otherwise it is

[9] The fog hint is not verified, so it can technically be used for any purpose.

considered 'dead' and is no longer eligible for adding to the chain.[10,11]

7.6 Transaction receipts

When sending an output to someone, it is useful to prove that you sent it. For that purpose, MobileCoin has so-called '`Receipts`'. Transaction authors can make a `Receipt` for each of the outputs they create.[12] The contents of a `Receipt` are fairly self-explanatory.

api/proto/external.proto `Receipt`

- **txout public key**: The txout public key of the output sent.
- **confirmation number**: Hash of the sender-receiver shared secret, $\mathcal{H}(r_t K_t^{v,i})$. By recomputing the confirmation number, recipients can gain confidence that whoever sent them the `Receipt` was the output creator (assuming the output actually exists).[13]

[MC-tx] src/tx.rs `TxOutConfirmationNumber` `validate()`

- **tombstone block**: Nominal tombstone block of the transaction that contains the output. This isn't verifiable, but helps the recipient know whether the transaction succeeded or failed. If the output they supposedly own doesn't show up on-chain before the tombstone block, then the transaction probably failed.
- **amount**: Nominal amount contained in the output. This can only be verified by looking at the real transaction output. It is included for informational purposes.

[10] In the initial version of MobileCoin, a transaction must be submitted to the network within 100 blocks of its tombstone block, otherwise it will be considered invalid. This requirement will likely inhibit applications of multisignatures [100] where transactions take a while to construct and it isn't possible to accurately/reliably estimate when they will be submitted to the network. Hopefully future versions of the MobileCoin protocol will change or remove this behavior.

[MC-tx] src/validation/validate.rs `validate_tombstone()`

[11] See Section 11.2.1 for a use-case for tombstone blocks.

[12] Colloquially you get a 'receipt' back after you *send* money to someone, rather than getting a receipt alongside money received. MobileCoin's `Receipt` would have been more aptly named `PaymentNotice`.

[13] A `Receipt` can be used to perform a Janus attack (recall Chapter 4 footnote 13). If the attacker knows one subaddress belonging to the victim (e.g. their 'normal' subaddress), and suspects that a different subaddress belongs to the same account, then they can create a `Receipt` based on the suspected subaddress. After sending the `Receipt` to the victim, if the victim indicates that the confirmation number was valid, then the attacker will know both subaddresses belong to the victim. The victim has no way to detect this attack, because the `Receipt`'s txout public key and confirmation number cannot be tied to any specific subaddress they own. To fix this attack, you could add a new field to `Receipts` called **txout base key**, which is simply $r_t G$. Then the victim can compute $k^{s,i} r_t G \stackrel{?}{=}$ **txout public key**, where $k^{s,i}$ is the private spend key of the subaddress this `Receipt` is *supposed* to be based on.

CHAPTER 8

SGX Secure Enclaves

As mentioned in previous chapters, all MobileCoin transactions are validated in *secure enclaves*. We claimed these enclaves are opaque boxes that can't be examined by even those who own them, making it possible to safely discard identifying information like ring signatures before appending transactions to the blockchain.

On the face of it, secure enclaves may seem preposterous. How can we be sure they validate transactions properly? If information must be passed into an enclave, can't we just read it before passing it in? How can a transaction author send their transaction to the network, and be sure only secure enclaves can read it? How could the entire blockchain be created from the output of these secure enclaves?

Perhaps some readers have heard about secure enclaves before, and realize they are not as robust as elliptic curve cryptography or hash functions. In the case of an enclave breach, will MobileCoin collapse under the weight of a mistaken security assumption?

This chapter aims to shed light on how secure enclaves work (specifically Intel's Software Guard Extensions [SGX] technology [21, 39]), while Chapter 9 discusses how enclaves fit into the broader picture of consensuating transactions and growing the MobileCoin blockchain.

8.1 An outline

Suppose you want to send data to a remote server and get back the results of some computations on it. Clearly, without any special techniques, the server's operator will be able to read the input

data, and can return anything they want back to the user. Secure enclaves enable remote users to send data that can't be examined by the enclave operator, and be sure the results they get back actually came from the intended algorithm [39].

8.1.1 Secure enclaves

A secure enclave is a sectioned-off part of a computer not visible to, or modifiable by, anyone after it has been set up [21]. Whatever an enclave does with data passed in is determined by the software it is initialized with.

To pass data into an enclave, the data owner encrypts it using a public key associated with that enclave. Once inside the enclave, the encrypted packets are decrypted and consumed by the enclave's software. Depending how that software was designed, its outputs may be encrypted for transmission to the original data owner or another secure enclave, or sent in plaintext outside the enclave for use by the local machine.

An enclave may also encrypt some data for local 'sealed' storage, which it can access and decrypt again at a later time.

At this point the data owner is still faced with two problems.

8.1.2 Chain of trust

First, how can a data owner be sure the public key they encrypt their data with is specifically a secure enclave's public key? Public key cryptography is free to use for everyone.[1]

It turns out secure enclaves can't exist without specially designed hardware. Aside from the various details that go into making secure enclaves secure at the machine level, such hardware is manufactured with two secret values 'baked in' (e.g. 128-bit integers). Each piece of hardware gets its own pair of secrets, one of which is known to the manufacturer and the other of which is not. These secret values are used to create all the cryptographic keys essential to securely setting up and running enclaves. As we will see, there is no way to extract the hardware secrets, since they are never directly exposed to enclave software. [39]

Through a process we discuss in Section 8.2.5 involving the hardware secrets, the manufacturer (e.g. Intel) cooperates with enclave-enabled machines to create 'Attestation keys' that can be used by enclaves on those machines to prove a piece of data was produced by a secure enclave. For instance, a public key that can encrypt messages only readable by an enclave could be attested to.

[1] Note that message/data encryption can be done by creating a Diffie-Hellman shared secret, breaking the message into chunks, and then XORing each chunk with a value (e.g. hash) computed based on the shared secret. The recipient learns your public key and encrypted message, and decrypts it by recomputing the mask values and XORing them with the encrypted chunks. MobileCoin's secure enclaves use a more sophisticated technique called AES [13] to process the message chunks, where key exchange for creating the AES 'cipher key' relies on the Noise Protocol Framework [141] (which involves several Diffie-Hellman exchanges between the communicating parties). A communication channel where both parties have the same private key (e.g. shared secret) is called *symmetric encryption* [39].

8.1.3 Remote attestation

The second problem faced by data owners is that secure enclaves may not be running the software they expect. If they send data to an enclave that just sends it right back out for the local operator to read, there would hardly be a point to using enclaves in the first place.

When a secure enclave is being 'initialized' (see Section 8.2.4), the software it is supposed to run is passed in (i.e. its compiled binary form). Part of the initialization procedure is to progressively hash that software, with the final output known as a 'measurement hash'. [39]

Once an enclave has been initialized, the software it runs can't be modified, so the measurement hash is a reliable representation of the enclave's software. The measurement hash is constructed automatically by secure CPU instructions, and is stored alongside that enclave in a data structure only accessible by other secure CPU instructions.

When an enclave-enabled machine uses its Attestation key to prove that a message was produced by an enclave, it includes that enclave's measurement hash with the proof. Anyone who validates the proof can be confident the message was produced by an enclave running the implied software, assuming they trust the enclave technology is not completely flawed and the enclave provider (Intel in our case) implemented each step of the process correctly and honestly.

Any person who wishes to interact with an enclave is responsible for verifying the measurement hash is correct, and the software it represents is well made. There are two general ways to do this.

1. Obtain the measurement hash from a trusted source, who is assumed to have already validated the software. For example, the software developer who made it.
2. Manually validate the software source code, and recreate the measurement hash. This only works if the software is deterministically reproducible [54], which means the binary executable created by compiling the source code can be reproduced byte-for-byte when recompiling it (on the same or a different machine). The measurement hash can be recreated by anyone with the binary, so it is relatively easy to verify (no need for special hardware).[2]

8.2 Filling in the gaps

So far we have sketched how secure enclaves work *in principle*. The rest of this chapter attempts to flesh out that sketch with theoretical and practical details pertinent to Intel's SGX secure enclave technology [88, 90]. Unfortunately enclaves are quite complicated, involve a lot of hardware-related implementation details, and have incomplete documentation [21], so it is not feasible for us to explore everything that could be explored (e.g. threading, interrupts/exceptions, page eviction, and

[2] See for example `sdk/sign_tool/SignTool/sign_tool.cpp measure_enclave()` from Intel's `linux-sgx` library [89].

ECALLs/OCALLs for transitioning execution between an enclave and its host process are not addressed). See [39] (especially chapter 5), [157], and [90] for advanced treatments of the subject, and any unanswered questions.

At this point, enclaves may still sound arcane and imprecise. In reality, they are quite similar to normal processes (e.g. applications, server code, etc.) that are ubiquitous on modern computers/servers. An enclave can be thought of as a 'dynamically loaded library' ([39] section 5.2) which is started up and then managed by a non-enclave host process throughout its lifetime.

Starting an enclave is analagous to starting any other process. A range of virtual memory ([39] section 2.5.1) is assigned to the process (called the `ELRANGE` for enclaves), and the process's intended executable code is loaded into that memory ([39] section 5.3.2). The main differences between enclaves and normal processes are how memory is initialized, memory access rights, and the existence of secure CPU instructions only usable in the context of an enclave.

8.2.1 SGX implementation

The implementation of SGX has four essential aspects: a so-called *Memory Encryption Engine*, hardware secrets, secure CPU instructions, and memory checks that enforce what memory software can access.

The presence of a Memory Encryption Engine (MEE) [80] in SGX-enabled hardware constitues the main deviation from normal Intel CPU design. It is a standalone module in the CPU's uncore, and exists to prevent physical attacks on memory (DRAM) allocated to secure enclaves ([39] section 6.1.2). Aside from hardware secrets, all other parts of the SGX technology are implemented in microcode ([39] section 6.1).[3,4]

SGX hardware secrets

SGX-enabled hardware include two random hardware secrets added during the manufacturing process.[5] One of them, the *provisioning secret*, is generated in a secure facility, and a copy is stored permanently by Intel. The second, so-called *seal secret*, is created in-place within the CPU during production, and is discarded during the manufacturing process so it is only known to the device.[6] ([93] section 2.2)

[3] Microcode is essentially processor firmware [53], meaning it is permanent or semi-permanent executable code embedded in the processor [6]. Its purpose is translating higher level CPU instructions into hardware-level CPU operations [163].

[4] By implementing most of SGX in microcode, it is easier for Intel (relative to a hardware-heavy implementation) to both design and update it ([39] section 6.1). Since SGX is so complex, it is prone to vulnerabilities [8] that can only be addressed with security updates.

[5] Specifically, the secrets are burned into e-fuses. It is likely the secrets are encrypted with a 'global wrapping logic key', another hardware secret that may be shared between different chips in the same production line, to make the e-fuse secrets harder to extract via physical analysis. The exact details are unknown or proprietary. ([39] section 6.6.2 and [40] section 2.6)

[6] Intel documentation [93] labels what [39] calls the provisioning and seal secrets, 'Root Provisioning Keys' (RPK) and 'Root Seal Keys' (RSK) respectively. We go along with the unofficial convention for legibility.

SGX CPU instructions

Most of the SGX implementation is represented by a set of novel CPU instructions. These instructions are responsible for setting up enclaves, enabling entry to and exit from the enclave during code execution, and accessing hardware secrets for various parts of the remote attestation process. We will explore a number of instructions throughout the remainder of this chapter. Keep in mind only SGX CPU instructions can access the hardware secrets.

SGX memory checks

Based on its name, it would seem like the Memory Encryption Engine should be able to protect memory allocated to secure enclaves from being accessed and read by other processes. However, as it is designed to guard against physical attacks, the MEE is not well suited to addressing software-related problems. Instead, memory access checks are added to the 'Page Miss Handler' so only the software running in an enclave can access the memory allocated to it. Enclave software can also freely access unprotected memory (i.e. memory not allocated to any enclave). Note that even an enclave's host application/process does not have access rights to the enclave's memory. ([39] section 6.2)

Aside from the `ELRANGE` memory which stores an enclave's code and data, there is also a separate ([39] section 5.3.2) *SGX Enclave Control Structure* (`SECS`) containing metadata about the enclave.[7] The `SECS` can only be read from or written to by the SGX CPU instructions. ([39] section 5.1.3)

8.2.2 SGX keys

The hardware secrets are exclusively used as inputs to a key derivation process involving various information about an enclave. There are five SGX key variants that can be created, with different preconditions and input sets.[8] These keys are essential components of SGX, which must be understood before looking deeper at how SGX works.

Key derivation

Each key variant has a pre-defined set of inputs that are selected out of a larger input list. The inputs are collected together and treated as 'key derivation material' for an AES key derivation

[7] An enclave's `SECS` is the first thing to be allocated when initializing an enclave, and the last to be destroyed when its life ends ([39] section 5.1.3). Its content includes the enclave's measurements (`MRENCLAVE` and `MRSIGNER`, Section 8.2.3), its attributes (bit flags that include the enclave's initialization state, whether debugging mode is on/off, and which architectural extensions were enabled when compiling the enclave executable code [[39] section 5.2.2]), the enclave's size (a power of 2) and base virtual address (the `SECS` is used by the SGX CPU instructions to locate the `ELRANGE`), its product ID (`ISVPRODID`), security version (`ISVSVN`), and a few more obscure items. See [90] section 37.7 for a complete coverage of `SECS` content.

[8] It is worth emphasizing again that only a few of the SGX CPU instructions can access the hardware secrets. In other words, SGX keys can only be created by those few instructions (there is one exception).

algorithm ([39] section 5.7.5). The algorithm's cipher key is based on the *Master Derivation Key*, which is a function of the provisioning secret and current SGX implementation's security version ([93] section 2.1).[9]

$$\textit{Master Derivation Key} = f(\textit{provisioning secret}, \texttt{CPUSVN})$$
$$\text{SGX\_key}_{\textit{variant\_type}} = \text{AES}[\ \textit{Master Derivation Key}\ ](\text{derivation material})$$

Table 8-1 summarizes the possible key derivation material ([39] section 5.7.5). Unless noted with a section reference, we will not discuss any of the table entries in-depth, since they are mostly implementation minutiae covered extensively by [39] and [90].

Table 8-1: Key Derivation Material

Input Type	Description
`KEYNAME`	A two-byte representation of the key variant name.
`SEAL_FUSES`	The hardware seal secret.
`KEYID`	A 32-byte random integer, ensuring new keys are always unique.
`MRENCLAVE`	An enclave measurement (Section 8.2.3).
`MRSIGNER`	A hash of the public key used to sign an enclave (Section 8.2.3).
`OWNEREPOCH`	A 16-byte random integer that can be reset whenever ownership of the hardware changes.[10]
`MASKEDATTRIBUTES`	A set of bit flags corresponding to attributes present in the enclave the key variant was created for. It is only a subset of that enclave's attributes.[11,12]
`CPUSVN`	Security version number (SVN) of the SGX implementation.[13,14]
`ISVSVN`	Security version number of an enclave's software, as designated by the enclave's author.[15]
`ISVPRODID`	Product ID of an enclave's software, as designated by the enclave's author.[16]

[9] The Master Derivation Key for SGX implementations with old security versions can be recreated by implementations with newer security versions. This makes it easy to transfer data encrypted with old SGX keys to new versions, while the reverse transfer from new to old is prohibited by version control in the implementation. ([93] section 2.1)

[10] Including the owner epoch in key derivation material means new owners of the hardware can't decrypt the old owners' secrets, and the relevant SGX key variants can't be used to track hardware between owners. ([39] section 5.7.5)

[11] SGX keys can only be created if the `INIT` flag is set and the `DEBUG` flag is not. ([39] section 5.7.5)

[12] Only requiring a subset of enclave attributes in key derivation material allows secret migration between enclaves with different combinations of optional attributes. ([39] section 5.7.5)

[13] The `CPUSVN` is supposedly a concatenation of security versions corresponding to different components of the SGX implementation. ([39] section 5.7.3)

[14] Although not stated explicitly, [93] sections 2.1-2.2 imply the `CPUSVN` is responsible for controlling the Master Derivation Key transformation function.

[15] Enclave secrets can be migrated to an enclave with the same or higher `ISVSVN`, but not lower. ([39] section 5.7.2)

[16] Since SGX keys depend on the enclave product ID, secrets may not pass between enclaves with different IDs even if all other aspects of their identity are the same (measurements and security version) ([39] section 5.7.5). This makes it possible to isolate enclaves with different IDs, although the utility of that is unclear.

Key variants

There are three SGX instructions that can create SGX keys. Instruction `EGETKEY` can create all five variants, with a few restrictions on the enclaves allowed to call it ([90] pg. 40-111). Instructions `EINIT` and `EREPORT` compute the 'Launch key' and 'Report key', respectively. Note that `EGETKEY` and `EREPORT` are only callable from within an enclave's executable code ([39] sections 5.7.5 and 5.8.1), while `EINIT` is only called during the process of initializing an enclave (see Section 8.2.4).

Table 8-2: SGX Key Variants

Key Variant Name	Description	`EGETKEY` Restrictions
Seal key	Intended for migrating secrets between different versions of an enclave with the same author ([39] section 5.7.5), or long-term storage of secrets ([93] section 2.3).	None
Report key	Used when an enclave makes a `Report` to prove to another local enclave they had access to a piece of data (see Section 8.2.6).	None
Provisioning key	Allows a Provisioning enclave created by Intel to prove to an Intel provisioning service it is running securely on SGX-enabled hardware (see Section 8.2.5).	The calling enclave's `PROVISIONKEY` attribute must be set.
Provisioning Seal key	Similar to the Seal key, it is mainly used to transfer the Attestation key between the Provisioning and Quoting enclaves created by Intel (see Section 8.2.5).	The calling enclave's `PROVISIONKEY` attribute must be set.
Launch key	Used to make an `EINIT Token` that indicates to the `EINIT` instruction that an enclave being initialized was approved by a valid Launch enclave (see Section 8.2.4).	The calling enclave's `EINITTOKEN` attribute must be set, and its `SECS.MRSIGNER` field must equal the key hash `IA32_SGXLEPUBKEYHASH` embedded in the SGX implementation (see Section 8.2.4).

We will point out the significance of the `EGETKEY` restrictions in future sections that also discuss how and when each key variant is used. The Seal key won't be explored further, since it's just a normal key used to encrypt secrets. Generally it is best to avoid storing secrets long-term due to the risk of enclave security breaches, although as we will see in Section 11.3.3, sealing secrets is useful in case a device is power-cycled (or an enclave's host process crashes).

Creating a key

Table 8-3 (based on table 3 of [93] and table 40-64 of [90]) details the key derivation material of each key variant when created by `EGETKEY` (except items that are present in all variants, such as the key name).[17] Table value 'Direct' corresponds with items that `EGETKEY` pulls directly from the enclave (e.g. hardware secrets, SGX implementation security version, stuff from the `SECS` structure), while 'Input' signifies items pulled from a 'key request' passed as input to `EGETKEY`.

Table 8-3: `EGETKEY` Key Derivation Material Per Key Variant

Key Variant Name	SEAL_FUSES	KEYID[18]	MRENCLAVE	MRSIGNER	OWNEREPOCH	MASKEDATTRIBUTES[19]	CPUSVN[20]	ISVSVN[21]	ISVPRODID
Seal key[22]	Direct	Input	Direct	Direct	Direct	Input	Input	Input	Direct
Report key[23]	Direct	Input	Direct	-	Direct	Direct	Direct	-	-
Provisioning key	-	-	-	Direct	-	Input	Input	Input	Direct
Provisioning Seal key	Direct	-	-	Direct	-	Input	Input	Input	Direct
Launch key[24]	Direct	Input	-	-	Direct	Input	Input	Input	Direct

[17] `EINIT` and `EREPORT` obtain their key derivation material with different methods, as noted.

[18] Intel documentation (table 3 of [93] and table 40-64 of [90]) implies `EGETKEY` generates `KEYID` for the Provisioning and Provisioning Seal keys (by stating the value is obtained directly). However, in ([39] section 5.8.2) the `KEYID` is left blank for Provisioning keys. Our interpretation is the Provisioning-type keys use a 'default' value for the `KEYID`, which could be a hard-coded value or simply 32 zero bytes. This idea is reinforced by how the Provisioning key is used during attestation (see [89] and Section 8.2.5), where the `KEYID` isn't transmitted to a third-party who needs to recompute Provisioning keys (they must know the value to use in advance, hence a default of some kind).

[19] For `MASKEDATTRIBUTES`, 'Input' means the field is set by bitwise AND between the enclave's `ATTRIBUTES` and an input `ATTRIBUTEMASK`, while 'Direct' means the field is set equal to the enclave's `ATTRIBUTES`.

[20] If the `CPUSVN` is a key request input, then it must be less than or equal to the current SGX implementation's security version. ([39] section 5.7.5)

[21] The `ISVSVN` is always input to `EGETKEY` (et al.) explicitly. Wherever used, it must be less than or equal to the security version of the enclave that calls `EGETKEY`. Incidentally, this implies Provisioning Seal keys can usually only encrypt secrets for enclaves that share an `ISVSVN` (the `MRENCLAVE` is left out, so the key can potentially be recreated by a different enclave). Those keys are mainly or even exclusively used to pass Attestation keys [Section 8.2.5] from the Provisioning enclave to the Quoting enclave, which are paired together by design, so this limitation makes sense.

[22] `EGETKEY` has an input called the `KEYPOLICY`, which lets the user decide if the `MRENCLAVE`, `MRSIGNER`, or `ISVPRODID` should be left out of the Seal key (i.e. set to zero). This input is also used by other keys for some different, more obscure details. See [90] pg. 40-111 through 40-118 and section 37.18.2.

[23] If a Report key is created by `EREPORT` for a target enclave, then the target enclave's `MRENVLAVE` and `ATTRIBUTES` are taken as inputs, and the `KEYID` is populated from a `CR_REPORT_KEYID` field randomly generated each time the SGX-enabled hardware is restarted ([90] pg. 40-123). The target enclave's `ATTRIBUTES` are used directly, not a subset of them. This design works because the key is being created for a specific enclave that currently exists on the same machine, rather than generically for an enclave that *might* exist on the current machine.

[24] The `EINIT` instruction also computes the Launch key, to validate `EINIT tokens`. It takes as input all parts of the Launch key's derivation material except the seal secret and owner epoch. This may seem strange, since it implies

Keys can only be reconstructed if all their key derivation material is available, so typically information that is passed as an input to `EGETKEY` is stored alongside the key produced. See [39] chapter 5 for all the minor details, especially for the Report and Launch keys produced by `EREPORT` and `EINIT`.

We will not justify the different sets of key derivation material, in the interest of space. For an example of why a piece of information may be neglected from the key derivation, take the Provisioning key. It is the only key that does not use the seal secret, because the key is designed to be reproducible by Intel's provisioning service (Section 8.2.5), which does not know that value. All other keys *do* have the seal secret because they are supposed to be reproducible by only SGX CPU instructions.

8.2.3 SGX measurements

Before diving into the meaty parts of SGX (Sections 8.2.4, 8.2.5, & 8.2.6), we should also briefly introduce the two measurements `MRENCLAVE` and `MRSIGNER` that were mentioned earlier in this chapter.

Measurement `MRENCLAVE`

The `MRENCLAVE` is a measurement of an enclave's content; specifically, the binary code it is always initialized with (plus a few details about how the enclave will be organized when set up). It is constructed with the 256-bit SHA-2 hash function (a.k.a. SHA-256) ([90] section 38.4.1.1).

SHA hash functions are known as *block hash functions*, which first 'initialize' their internal state with initial values, then consume message chunks/blocks to 'extend' the hashed message, and at the end 'finalize' the hash by translating their internal state into a hash output. The hash function SHA-256 breaks the message-to-be-hashed into 64-byte blocks, has a 32-byte internal state, and outputs a 32-byte digest. ([39] section 3.1.3)

Since the measurement hash just depends on an enclave's initial content and SHA-256 (see Section 8.2.4), it can be easily computed without relying on any SGX CPU instructions (recall footnote 2).

Measurement `MRSIGNER`

Enclave authors provide a so-called `SIGSTRUCT` to potential users of an enclave. It contains the enclave's `MRENCLAVE`, miscellaneous metadata (like attributes the enclave is allowed to have and its security version), and the author's signature on that information (using the RFC 3447 [95] RSA signature scheme [[39] section 5.7.1]).[25] Users must pass that `SIGSTRUCT` (laid out in Table 8-4 [[39] section 5.9.1]) as an input when initializing the corresponding enclave.

---

any enclave could make an `EINIT token` that would be considered valid by `EINIT`. As we will see (Section 8.2.4), since `EGETKEY` restricts which enclaves can make Launch keys, construction of `EINIT tokens` is also restricted. By extension, this enables the `EINIT` instruction to filter out 'improperly' launched enclaves.

[25] This kind of message/signature combination is known as a 'certificate' (see [39] section 3.2).

Table 8-4: `SIGSTRUCT` Contents

Struct Field	Description
`ATTRIBUTES`	Attributes the enclave must have when instantiated.
`ATTRIBUTEMASK`	The `SIGSTRUCT`'s attributes must equal the bitwise AND between instantiated enclaves' real attributes and the attribute mask.[26]
`VENDOR`	Indicates if the enclave was produced by Intel. Typically '0' if not. It's not clear if this offers any utility.
`DATE`	Defined by the enclave author (e.g. the struct's signer). Possibly allows users to identify how old an enclave is.
`ENCLAVEHASH`	Expected `MRENCLAVE` measurement of the enclave.
`ISVPRODID`	Enclave product ID defined by enclave author. Used to populate the corresponding `SECS` field ([39] section 5.7.4).
`ISVSVN`	Enclave security version defined by enclave author. Used to populate the corresponding `SECS` field.
RSA Signature	An RSA signature on the rest of the `SIGSTRUCT`, containing {exponent, modulus, signature, Q1, Q2}.[27] The signature's 'modulus' constitutes the signer's public key, and its SHA-256 hash is the `MRSIGNER` value.

During enclave initialization (Section 8.2.4), the `SIGSTRUCT`'s signature is verified, and the `MRSIGNER` field in the `SECS` structure is set to the SHA-256 hash of the signature's public key. Since the `SECS` structure (mentioned in Section 8.2.1) can only be accessed by SGX CPU instructions, the `MRSIGNER` is sufficient proof post-initialization of who authored an enclave.

The `MRSIGNER` is required for identity-based privileged attribute checks in `EINIT` and the Launch enclave (Section 8.2.4). More generally, it allows an enclave author to safely migrate secrets between versions of their enclaves, since `MRSIGNER` can be part of the key derivation material for Seal keys ([90] section 38.4.1.2). If there was no `MRSIGNER`, then anyone could create a 'new' version of a given enclave and trivially expose any secrets the original enclave had sealed.[28]

[26] In more detail, the `SIGSTRUCT`'s attributes and attribute mask work together to prevent instantiated enclaves from having disallowed attributes, while leaving room for optional and required attributes. There are three useful attribute/mask bit combinations. If both are true, then the attribute is required. If just the attribute bit is false, then the attribute is disallowed (real attribute & mask bit must equal zero, only possible if real attribute isn't present). If both are false, then the attribute is optional (whether or not the real attribute is set, will have no affect on the attribute test). If just the mask is false, then it is impossible for the test to work, and the enclave will always be rejected (hence it's not a useful combination).

[27] The terms Q1 and Q2 in the `SIGSTRUCT` RSA signature are temporary values that make verifying it easier, likewise with the exponent which is the integer '3'. ([39] section 6.5)

[28] Even adding the `MRSIGNER` restriction to Seal keys, allowing secret migration between enclave versions is still dangerous. The enclave author's private keys could become compromised, or the author himself may be malicious. MobileCoin does not allow any enclave secrets to persist between enclave versions, in part due to these dangers.

8.2.4 SGX enclave initializing

Initializing an SGX enclave takes four basic steps. After it is initialized, the enclave's executable binary is static, and only that binary is able to read and edit the enclave's internal malleable state.

Enclaves are loaded in 4 KB units, called 'Enclave Page Cache (EPC) pages', to harmonize with the 4 KB page size of the address translation feature in Intel's CPU architecture ([39] section 5.1.1). To ensure all enclaves are set up the same, enclave authors must specify the enclave 'memory layout'. In other words, what initial content should be loaded into which pages within the `ELRANGE`, and what access restrictions those pages should have (i.e. a page with executable code, or a page readable/writable by enclave code [[39] section 5.2.3]).

Step 1: Assigning virtual memory

An enclave's life starts with the `ECREATE` SGX CPU instruction.

- `ECREATE`: A free EPC page is commandeered to be the enclave's `SECS`, and initialized. In the `SECS`, there is an `ATTRIBUTES` field containing all the enclave's attributes. One such attribute is the `INIT` flag, which starts out false. Only when `INIT` is false can data be added to the initial enclave state. Presumably the `ELRANGE` is also set up by this instruction. ([39] section 5.3.1)

  To reiterate, the `ELRANGE` is a range of virtual memory (composed of EPC pages) which, once assigned, can only be modified by SGX CPU instructions. The enclave's `SECS` is an EPC page containing metadata about the enclave exclusively accessible and modifiable by SGX CPU instructions throughout its lifetime, which does not reside within the `ELRANGE`.

  An enclave's measurement begins here as well. The SHA-256 algorithm is initialized and extended with the first 64-byte block. That block is composed of: {"ECREATE" (i.e. the string in byte form), SSA frame size ('State Save Area' for saving state when an exception is encountered that forces the processor to leave the enclave [[39] section 5.2.5]), `ELRANGE` size (number of EPC pages allocated to the enclave), padding}.[29] ([39] section 5.6.1)

Step 2: Loading the enclave

Next, the enclave executable code and initial data are loaded into the `ELRANGE` by SGX instructions `EADD` and `EEXTEND`, which also extend the enclave measurement hash based on the bytes loaded in. These instructions only work if the `INIT` flag is false.

[29] Enclave attributes are not included anywhere in the measurement hash, in case the enclave author wants a single measurement to be valid for multiple attribute permutations. An enclave's attributes include the enclave extensions it is allowed to use, but multiple combinations of extensions may be considered valid. Instead, attribute information is included with the attestation signature, so remote users can verify if the attributes are within a legitimate range of possibilities ([39] section 5.6.2).

- `EADD`: To load a new non-enclave page into the `ELRANGE`, its location in memory is connected with the intended EPC page virtual address in a `PAGEINFO` structure by `EADD`. That structure also references the enclave's `SECS` and the new page's access restrictions (readable/writable/executable). It can be thought of as a metadata 'header' for preparing a new page to be loaded. ([39] section 5.3.2)

  Once a new page is prepared, it is copied from non-enclave memory into the appropriate EPC page by `EADD`. ([90] section 38.1.2)

  Each `EADD` instruction extends the measurement hash with a 64-byte block composed of {"EADD", expected virtual address of EPC page to be added, `PAGEINFO` containing {enclave page type (it can be a regular code/data page, or a special 'Thread Control Structure' for dealing with multithreaded enclaves), access permissions}}.[30] ([39] section 5.6.3)

- `EEXTEND`: After an EPC page has been initialized by `EADD`, its contents are divided into 256-byte chunks for updating the measurement hash. This implies it takes 16 `EEXTEND` instructions to fully measure a 4 KB page. ([90] section 38.1.2)

  Each `EEXTEND` instruction extends the measurement hash with five 64-byte blocks. The first block contains {"EEXTEND", enclave offset (i.e. relative location of the 256-byte chunk within the `ELRANGE` [[90] pg. 40-45][31]), padding}, and the remaining four blocks divide up the 256-byte chunk. ([39] section 5.6.4)

Step 3: `EINIT token` (and the Launch enclave)

In Table 8-2, we indicated Provisioning and Provisioning Seal keys can only be produced by `EGETKEY` when the enclave that calls it has the `PROVISIONKEY` attribute set. Without further restrictions, that attribute check would be fairly pointless, as any enclave author could set the flag arbitrarily.

It is useful to restrict which enclaves can make these keys because there is no `OWNEREPOCH` in their key derivation material (recall Table 8-3). To recap, the owner epoch is a random value stored in an Intel machine that can be reset whenever ownership of that machine changes (Table 8-1). Keys derived without the owner epoch allow a malicious enclave to 'track' a piece of hardware across ownership changes, and let new owners decrypt secrets previously encrypted with those keys. ([39] section 5.7.5)

Therefore, it is beneficial to owners of SGX-enabled hardware to limit which enclaves may have the `PROVISIONKEY` attribute. As [39] points out in section 5.9.3, it is technically possible to implement a launch control policy for enclaves in system software. Such a policy would prevent an enclave with the `PROVISIONKEY` attribute set from being initialized/launched if its contents (`MRENCLAVE`)

[30] Forcing the measurement hash to be dependent on the memory layout and access permissions of each page ensures the main part of each enclave is fully deterministic when initialized by different enclave operators.

[31] The precise nature of the enclave offset is not clearly stated in Intel's documentation, however the sample code on page 40-45 of [90] implies it is the relative location of the 256-byte chunk within the `ELRANGE`, rather than the relative location of the page itself as described by [39] sections 5.3.2 and 5.6.4.

or author (MRSIGNER) are not in a pre-approved list. Unfortunately, Intel does not allow owners of SGX-enabled hardware to implement a launch control policy purely ad hoc. Instead, the policy must be implemented in a so-called 'Launch enclave', which is required for enclave initialization.

A launch enclave takes as input an enclave-being-launched's SIGSTRUCT and its intended attributes. It outputs an EINIT token signifying it approves of the enclave being launched. This token is a required input to the EINIT CPU instruction (step 4, discussed next), and is constructed based on a Launch key made by EGETKEY. Furthermore, as we will see, only a Launch enclave provided (i.e. signed) by Intel is allowed to make Launch keys via EGETKEY. The net effect of this is enclaves can only be initialized based on the approval of a Launch enclave obtained from Intel.[32] Table 8-5 details the content of an EINIT token.

Table 8-5: EINIT token Contents

Token Field	Description
MAC	Basically a hash of the next four token fields (ATTRIBUTES thru MRSIGNER) and a Launch key (recall Section 8.2.2) created by calling EGETKEY from the Launch enclave.[33]
ATTRIBUTES	The enclave-being-launched's intended attributes (an input to the Launch enclave).
VALID	A true/false flag indicating if the Launch enclave considers the enclave-being-launched valid or invalid (i.e. the launch control policy's verdict).
MRENCLAVE	Measurement of the enclave-being-launched (from its SIGSTRUCT).
MRSIGNER	SHA-256 hash of the public key used to sign the enclave-being-launched (from its SIGSTRUCT).
ISVSVNLE	The Launch enclave's security version. For recreating the Launch key.
KEYID	A random number used to make the Launch key unique. For recreating the Launch key.
CPUSVNLE	The SGX implementation's security version, as used to make the Launch key. For recreating the Launch key.
ISVPRODIDLE	The Launch enclave's product ID. For recreating the Launch key.
MASKEDATTRIBUTESLE	A subset of the Launch enclave's attributes. For recreating the Launch key.

Only if the SIGSTRUCT is signed with an Intel public key will the Launch enclave allow the PROVISIONKEY attribute to be set in the enclave-being-launched's intended attributes ([7], [39] section 5.9.1).[34] That is the entirety of the launch control policy as far as the documentation

[32] On a subset of SGX-enabled hardware, it is possible to set up 'Flexible Launch Control' [94] and implement a custom Launch enclave [7]. Based on [7], it seems FLC-type Launch enclaves do not require approval from Intel. We may investigate FLC and related techniques [149] for reducing Intel's role in using SGX enclaves in future editions of this report.

[33] The EINIT token's MAC is produced by the AES-CMAC algorithm. ([39] section 5.9.1)

[34] It is not made clear in [39] or [90] if the Launch enclave contains a hard-coded Intel key (or set of keys), or

reveals. In practice, only Intel's Provisioning enclave (Section 8.2.5) and Quoting enclave (Section 8.2.6) are allowed to have the `PROVISIONKEY` attribute.

Step 4: `EINIT`

Finally, the `EINIT token` and `SIGSTRUCT` are passed to the `EINIT` instruction, which finalizes the measurement hash, verifies the `EINIT token` and `SIGSTRUCT` signature, ensures the enclave doesn't have any attributes not permitted by the `SIGSTRUCT`, sets several values in the `SECS` structure based on the `SIGSTRUCT`, and marks the enclave as initialized to prevent any further modifications. It is instructive to directly enumerate the steps taken, as described by Intel's SGX developer manual ([90] pg. 40-47 to 40-52; the exact order of steps may not fully match the SGX implementation, and some trivial or obscure steps are left out).

1. Complete the enclave's measurement by finalizing the SHA-256 algorithm (translating the algorithm's internal state into a hash output). The final value is stored in the enclave's `SECS.MRENCLAVE` field.[35]
2. `SIGSTRUCT` checks:
   (a) Verify the `SIGSTRUCT`'s signature.
   (b) Check if the `SIGSTRUCT`'s `ENCLAVEHASH` matches the final `MRENCLAVE` value.
   (c) Check if the enclave's real attributes are permitted by its author. Bitwise AND between `SECS.ATTRIBUTES` and `SIGSTRUCT.MASKEDATTRIBUTES` must equal `SIGSTRUCT.ATTRIBUTES`.
3. Set the `SECS` fields `SECS.MRSIGNER`, `SECS.ISVPRODID`, and `SECS.ISVSVN` by copying the corresponding values from the `SIGSTRUCT` (recall the `MRSIGNER` is computed as a SHA-256 hash of the enclave author's public key, as found in the `SIGSTRUCT`).
4. Check if the `LAUNCHKEY` attribute is not set in `SECS.ATTRIBUTES` unless the enclave's `MRSIGNER` equals a value known as `IA32_SGXLEPUBKEYHASH` (a.k.a. 'SGX Launch enclave public key hash'), which is embedded in the SGX implementation.
5. `EINIT token` checks:

if it somehow uses the `IA32_SGXLEPUBKEYHASH` value embedded in the SGX implementation to check `SIGSTRUCT` signer keys. It is also not clear if the launch control policy is applied to the `SIGSTRUCT` attributes (which are constraints on what the real attributes may be) or the intended attributes. The FLC-type reference Launch enclave implementation at [89] `/psw/ae/ref_le/ref_le.cpp ref_le_get_launch_token()` implies it is the intended attributes that are controlled. Moreover, the `EINIT token MAC` uses the intended attributes directly, implying they have a central role in the launch control policy. These are implementation details and ultimately don't affect the observable behavior of the Launch enclave, especially since the `EINIT` instruction is quite rigorous about checking for attribute inconsistencies.

[35] In fact, during initialization the intermediate measurement hash values are also stored in the `SECS.MRENCLAVE` field for convenience and to isolate them from unauthorized modification (only the SGX CPU instructions can read from, and write to, the `SECS` structure). ([39] section 5.6)

(a) If the token is marked invalid, make sure the `SECS.MRSIGNER` value equals `IA32_SGXLE-PUBKEYHASH`, then skip the other token checks.

(b) Use the token contents to compute the Launch key and verify the token's `MAC`. This computation is integrated directly into `EINIT`, and is a sort of duplicate implementation of `EGETKEY:LAUNCHKEY`.

(c) Verify the token's `MRSIGNER` value matches the `SECS.MRSIGNER` value. Also check the corresponding `MRENCLAVE` values.

(d) Check if the token's 'intended attributes' field matches the enclave's real attributes (`SECS.ATTRIBUTES`).

6. If all checks pass, mark the enclave as initialized by setting the `SECS.INIT` flag.

The reader may have noticed we snuck a crucial design detail into the list. Enclaves signed by Intel's public key, which is hard coded into Intel's SGX implementation, can get past `EINIT` without a valid `EINIT token`. This provides a path to 'bootstrap' enclave launching. First launch Intel's Launch enclave, then use it to make `EINIT token`s for launching all other enclaves.

8.2.5 SGX provisioning

To achieve untrusted remote computation, secure enclave owners must prove they are running enclaves with software expected by remote parties. It isn't enough that the *owners* know they are running secure enclaves. Since secure enclaves are ultimately a processor technology, remote computation proofs must be connected to the relevant processor manufacturer (e.g. Intel).

In short, the enclave-enabled hardware proves knowledge of the provisioning secret to its manufacturer (who also knows that secret), the manufacturer provisions an 'Attestation key' to the hardware signifying it trusts the hardware, enclaves on that machine use the Attestation key to sign their messages, the manufacturer verifies those signatures and filters out compromised signers (a capability of the EPID signing protocol [34]) before re-signing it, and final recipients of messages check the manufacturer's signatures against a trusted public key. We discuss provisioning in this section, and attestation in the next.

Provisioning enclave

Provisioning is orchestrated by a 'Provisioning enclave' signed by Intel.[36] This is one of the enclaves allowed by the Launch enclave to have the `PROVISIONKEY` attribute (Section 8.2.4, step

[36] According to [91] and [161], provisioning is designed by Intel to use two enclaves (the 'Provisioning enclave' and 'Provisioning Certification enclave'), however [17] claims (without direct citation) these are implemented in practice as one combined enclave. It seems at least Intel's Linux implementation uses separate enclaves [89]. For simplicity, we assume there is only one enclave. The main purpose of splitting out the certification enclave appears to be encapsulating a check on the `PROVISIONKEY` attribute, such that the certification enclave will only produce the PPID for authorized enclaves, and also to provide a rigorous process for 'certifying' that an enclave is so authorized [89].

3). We recommend [91] section 2.4.5.1, [161], and especially [89] for a more complete picture of the provisioning process than is laid out here.

Intel obtains a copy of a hardware's provisioning secret during the manufacturing process. Moreover, that hardware can only use the provisioning secret to create SGX keys, of which only the Provisioning key can be made by Intel (all other keys use the seal secret). Therefore, for a piece of hardware to prove it is SGX-enabled, it must create a Provisioning key that Intel can verify was created by a provisioning secret stored in their database.

However, how can Intel know which provisioning secret to use, to recreate a given Provisioning key it is presented with? It is unreasonable to test *all* secrets in its database, so Intel must use an identifier of some kind to locate the relevant provisioning secret.

Intel's solution has two steps:

1. The Provisioning enclave uses `EGETKEY` to produce a Provisioning key, with the `CPUSVN` and `ISUSVN` values explicitly set to zero. A hash of that key, called the Provisioning ID (PPID), is sent to Intel (in encrypted form, as is standard for network communications). Intel can precompute the PPID for all stored provisioning secrets, making it easy to connect provisioning requests with secrets created during manufacturing.[37]

2. The Provisioning enclave produces a second Provisioning key, this time with the real `CPUSVN` and `ISUSVN` values, and sends it to Intel. Intel can reproduce this new key using the provisioning secret associated with the previous step's PPID, along with the security versions that are also transmitted in this step. Intel can block provisioning requests from machines with insecure SGX or Provisioning enclave implementations.

In practice [161], the second step could be merged with the first, or woven into the EPID Join protocol which is discussed next.

EPID Join protocol

Rather than give an Attestation key directly to the SGX-enabled hardware, Intel's provisioning service collaborates with the Provisioning enclave on an 'EPID Join protocol' ([34], [91] section 2.4.5.1, [161]). At the end of this protocol, the Provisioning enclave will have a private Attestation key unknown to Intel, which can be used in the remote attestation process (see Section 8.2.6).

Whenever an SGX-enabled hardware goes through provisioning, Intel stores a copy of the PPID used and an encrypted version of the Attestation key created. It is encrypted with the Provisioning

[37] The SGX implementation and enclave security versions are left out of the PPID to ensure it is a value that will remain constant forever. Note that, per Table 8-3, the `MRSIGNER` and `ISVPRODID` are still a part of the PPID, so in practice only Provisioning enclaves signed by Intel can create useful PPIDs.

Seal by the hardware (which only the hardware can decrypt), allowing secure key recovery at a later date ([91] section 2.4.5.1).[38]

Attestation keys become 'members' of a group public key for the EPID (Enhanced Privacy ID) signature scheme [34] after 'joining' the group. The EPID Join protocol [34] requires passing three messages between the group key 'issuer' (Intel) and the 'platform' (SGX-enabled hardware).[39]

1. The issuer sends their EPID group key material to the platform.[40]
2. The platform generates part of their EPID private key, uses it to manipulate the group key material, and sends that manipulated material back to the issuer as a 'challenge'.
3. The issuer 'responds' with further manipulation of that material, which the platform uses to complete their EPID private key. Note that the group key material modifications are only temporary, and don't persist beyond the Join protocol.

The Attestation key (EPID private key) can be used to make signatures, which Intel can validate with the relevant EPID group key. Interestingly, an Attestation key's signatures are detached (as far as Intel knows) from the hardware that provisioned it. This provides increased privacy protections for operators of SGX enclaves ([39] section 5.8.2).[41] EPID permits the issuer (Intel) to reject attestation signatures produced by compromised group members [34, 93].[42]

Pass Attestation key to Quoting enclave

Since the Attestation key is the linchpin of remote attestation, it should only be accessible by carefully designed enclaves. In fact, only one enclave is intended to use it, namely the Intel-provided Quoting enclave. This enclave is in charge of the remote attestation process (discussed next), and is the second enclave authorized by the Launch enclave to have the `PROVISIONKEY` attribute (Section 8.2.4). To transmit the Attestation key from the Provisioning to the Quoting enclave, it is encrypted with a Provisioning Seal key and stored by the system software for later decryption and use by the Quoting enclave ([39] section 5.8.2).[43]

[38] During secure key recovery, or re-provisioning a new Attestation key (e.g. following an SGX implementation update), the hardware must sign a message with the old Attestation key (e.g. after obtaining and decrypting the encrypted version) and pass it to Intel ([91] section 2.4.5.1). This requirement allows Intel to check if the old key was revoked, and prevent re-provisioning requests to hardware that may be compromised [93].

[39] If Intel already has a copy of the provided PPID, then it will skip the EPID Join protocol and just send back the relevant encrypted Attestation key stored in anticipation of key recovery requests ([91] section 2.4.5.1). Of course, if the hardware operator intends to re-provision a new key, then the Join protocol will be enacted.

[40] Unlike Ed25519 curve points, which can be represented with a single value (once compressed), EPID keys (both public and private) are composed of multiple distinct values [34].

[41] EPID signatures can be linkable or unlinkable (i.e. one signature linkable to another signature), depending on the hardware operator's preference [93, 21]. MobileCoin nodes and Fog service operators currently use the linkable variant by default [128], which makes it easier to identify and revoke compromised Attestation keys [93]. EPID linkability may become mandatory in future versions of MobileCoin, according to private correspondence with MobileCoin developers.

sgx/epid-types/src/quote_sign.rs `quote_sign()`

[42] We leave the intricacies of EPID and its revocation mechanism to the paper [34] where it was first introduced.

[43] It may seem like Attestation keys can only be safely encrypted with the Provisioning Seal key thanks to the `PROVISIONKEY` limitation. However, it is actually the `MRSIGNER` input to Provisioning Seal keys that secures

8.2.6 SGX attestation

To interact remotely with an enclave, usually the first step is to open a secure communication channel with it. Of course, a channel should only open successfully if the remote enclave is really an enclave. To that end, the key exchange process between remote enclave and would-be user (e.g. as part of the TLS [9] or Noise [141] communication protocols)[44] should include a proof that the enclave is, in fact, an enclave.

However, the enclave that is setting up a communication channel isn't equipped to create that 'attestation' proof by itself. As we have discussed, only the Quoting enclave is able to produce signatures with the Attestation key. Instead, the enclave creates a local `Report` proving to the Quoting enclave it is a secure enclave on the same machine. The `Report` attests to a specific message (i.e. a `Report` says "this message was produced by a legitimate SGX secure enclave").

The Quoting enclave verifies the `Report`, then signs its included message with the Attestation key, producing a so-called `Quote`. This `Quote` can be transmitted to Intel, which uses the relevant EPID group key to verify the attestation signature (or reject it if produced by a revoked signer), then re-signs the message with an Intel RSA key. The Intel-signed message is known as an `Attestation Verification Report`, and gets sent back to the enclave operator, who can forward it to the original communication partner. That partner verifies the Intel signature and uses the message contents to complete the communication channel set-up.

Once a channel is established, there is no need for further attestation. Moreover, enclaves only need one public key to set up a communication channel with arbitrary numbers of users. An enclave can produce a single `Attestation Verification Report` containing that public key, and store it indefinitely in anticipation of requests from users who want to communicate with the enclave.

SGX local attestation

Attestation between enclaves on the same machine relies on the `EREPORT` SGX CPU instruction ([39] section 5.8.1). This instruction takes as input a message to report on (`REPORTDATA`), and the `MRENCLAVE` and attributes of the target enclave (the enclave being reported to). It outputs a `Report`. If used in the context of remote attestation, the target will be the Quoting enclave.

Internally, `EREPORT` makes a Report key on behalf of the target enclave (using its measurement and attributes instead of the calling enclave's values; recall Section 8.2.2). The Report key is used as the cipher key to an AES-CMAC algorithm, which basically outputs a hash (called a `MAC`) of inputs that is dependent on the cipher key (similar to how the `EINIT token` is constructed).

Attestation keys. Only a Quoting enclave produced by the same enclave author as the corresponding Provisioning enclave may decrypt Attestation keys, and only a Provisioning enclave approved by the manufacturer may obtain useful Attestation keys. In this way, the manufacturer can ensure Attestation keys are safely managed throughout their lifetimes.

[44] MobileCoin nodes are designed to use the Noise protocol, as implemented in the `mc-crypto-noise` Rust crate.

The AES-CMAC inputs are identifying details about the reporting enclave (obtained directly from its SECS), and the Report's message. When a target enclave receives a Report, it can use EGETKEY to recreate the Report key, then recompute the MAC to verify the Report's stored value. If the MACs match, then the Report's message must have been produced by an enclave (specifically, the enclave represented by the hashed information) on the same machine as the target enclave.[45]

Table 8-6 itemizes the content of a Report (see [39] section 5.8.1, or [86] section 4.3.1).

Table 8-6: Report Contents

Fields	Description
MAC	The output of an AES-CMAC algorithm (more-or-less a hash of inputs), with the Report key as cipher key. All other fields in the Report (aside from the KEYID) are inputs to the algorithm.
ATTRIBUTES	Attributes of the enclave that called EREPORT.
MRENCLAVE	Measurement of the enclave that called EREPORT (Section 8.2.3).
MRSIGNER	Hash of the public key used to sign the enclave that called EREPORT (Section 8.2.3).
ISVPRODID	Product ID of the enclave that called EREPORT.
ISVSVN	Security version number of the enclave that called EREPORT.
CPUSVN	Current security version number (SVN) of the SGX implementation.[46]
REPORTDATA	Message provided as an input to EREPORT.
KEYID	A random number used to make the Report key unique. Allows the Report key to be recreated, but does not otherwise contribute to making the MAC.

SGX remote attestation

After the Quoting enclave receives a Report containing a message intended for remote attestation (and validates it), there is not much left to say [86].

1. The Quoting enclave replaces the Report's MAC with an EPID signature using the Attestation key ([39] section 5.8.2), transforming it into a Quote.
2. The Quote is sent to Intel's Attestation Service (IAS), which verifies the EPID signature and checks if the signer was revoked.[47] Intel re-signs the Quote with an Intel RSA key, transforming it into an Attestation Verification Report (AVR) [86].

[45] Only the EREPORT SGX CPU instruction is able to make Report keys on behalf of another enclave. This means the target enclave can be sure a valid Report was definitely made by a secure enclave, as only secure enclaves may invoke EREPORT.

[46] Unlike other SGX key variants which take the CPUSVN as an input (Table 8-1), Report keys are generated (by both EREPORT and EGETKEY) using the current SGX implementation's security version. Doing so ensures outstanding Reports become worthless after every SGX implementation security update ([39] section 5.8.1).

[47] Intel can periodically revoke entire EPID groups in parallel with an SGX security update (inferred from the 'isvEnclaveQuoteStatus::GROUP_REVOKED' status that can be returned in AVR requests [86]). This forces all

3. The `AVR` is sent back to the enclave operator, who may forward it to a desired enclave user.

4. The user verifies the `AVR` signature and checks that the signing public key belongs to Intel.[48] If the `AVR`'s contents (measurements, attributes, etc.) meet the user's expectations, they can continue communicating with the enclave.

attest/core/src/ias/verify.rs `VerificationReport::verify()`

members of that group to re-provision new Attestation keys if they want to obtain new `AVR`s. Forced re-provisioning prevents Attestation keys that were exposed to potentially insecure SGX implementations from being used to sign `Quote`s with new `CPUSVN`s. This incidentally provides a convenient mechanism for enforcing SGX security updates in MobileCoin. If Intel issues an SGX update that triggers re-provisioning on new attestation requests, then MobileCoin can hard fork (author a new validator enclave with new `MRENCLAVE`) to force nodes to update their SGX implementations. Nodes contain enclaves that securely send encrypted transactions to each other, but the enclaves can only peer with enclaves that have the same `MRENCLAVE` value (based on the `AVR`s they receive from those nodes). Since these `AVR`s can only be made after the new enclave is released, if the enclave is released after a re-provisioning-inducing update, the `AVR`s must contain `CPUSVN` values that are considered secure by Intel. Without hard forking in this way, nodes can store old `AVR`s made for old SGX implementations and continue using them to establish communication channels with other enclaves and users even when Intel would force a re-provisioning if the nodes were to obtain new `AVR`s.

*crypto/ake/enclave/src/lib.rs `get_verifier()`

[48] In reality, `Attestation Verification Report`s include a certificate signing chain, which extends from the signing key to a root 'Attestation Report Signing CA Certificate' that should be trusted by the user. [86]

attest/core/src/ias/verify.rs `VerificationReport::verify_signature()`

CHAPTER 9

MobileCoin 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 [125], and in the digital realm we can offer our productivity to the whole world.

Media of exchange (moneys) are essential, facilitating economic calculation with respect 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 [125]. 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. [130]

To alleviate those drawbacks, decentralized digital currencies have been engineered.

9.1 Digital currency

At the most basic level, a digital currency is just a collection of messages, and the 'amounts' recorded in those messages are interpreted as monetary quantities. There are three types of digital currencies: personal, centralized, or distributed.

In the **email model**, anyone can make coins (e.g. a message saying "I own 5 coins"), and anyone can 'send their coins' to whoever has an email address. There is no limit to the supply of coins, nor is there a way to prevent someone from 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.

9.1.1 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 for goods and services, 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.[1]

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 7, 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.[2] 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 distributed consensus protocols and blockchain technology as found in cryptocurrencies.

[1] In political science, this is called a Schelling point [74], social minima, or social contract.
[2] In commodity money like gold, these rules are met by physical reality.

9.1.2 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 an 'origin' (or 'genesis', as it is more commonly known) declaration: "Let the SampleCoin begin!". We call this message a 'block', and its block hash is

$$BH_O = \mathcal{H}(\text{"Let the SampleCoin begin!"})$$

Every time a node receives some transactions, they use representations (e.g. hashes or bit-strings) of those transactions like messages, along with the previous block's hash, and compute new block hashes

$$BH_1 = \mathcal{H}(BH_O, TX_1, TX_2, ...)$$
$$BH_2 = \mathcal{H}(BH_1, TX_3, TX_4, ...)$$

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 origin message. We have a very simple 'blockchain'.[3]

9.2 Consensus protocol intro

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). Moreover, the likelihood of this happening rises with the frequency blocks are published.

In 'classical' cryptocurrencies modeled after Bitcoin, the speed at which blocks are produced, and the mechanism for resolving forks, is governed by the *Nakamoto consensus protocol* [130].[4] That protocol has some drawbacks, which include susceptibility to 51% attacks that roll back transaction events (especially for cryptocurrencies where the fiat-denominated cost of those attacks is relatively low), the energy cost of proof-of-work algorithms is worrisome to those concerned about the environment or sustainability, and it can take on the order of tens of minutes to confirm that a transaction has been successfully added to the blockchain (as opposed to mere seconds for e.g. credit card transactions) [108].

[3] 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. [3]

[4] At the heart of the Nakamoto consensus protocol, nodes compete for the right to publish new blocks by randomly generating numbers (nonces) that get inserted into blocks, until the block hash (using an inefficient 'proof-of-work' hash algorithm) meets some condition. For example, if the hash value times a 'difficulty' value is less than the maximum hash value, such that higher difficulties require more unlikely hash outputs, then a block can be published with the relevant nonce. By controlling the difficulty value dynamically, it is possible to control the speed at which blocks are produced, and chains with the highest cumulative difficulty (over all blocks) are considered canonical in fork scenarios. Since difficulty translates to work done computing proof-of-work hashes over time, chains with the highest cumulative difficulty are considered most likely to represent a network consensus. [100]

9.2.1 Stellar Consensus Protocol

MobileCoin obtains consensus on blockchain contents with a different model known as *federated Byzantine agreement*, specifically an implementation of the Stellar Consensus Protocol (SCP) which is based on that model [115]. In SCP, before the blockchain can be extended with a new block, all (or at least all well-behaved and well-connected) nodes must come to an agreement about which block should be added next. This allows SCP-based blockchains to add new blocks quite rapidly (on the order of seconds in the typical case [108]), because automatically reaching consensus on new blocks can occur at the speed of Internet connections between nodes.

In Chapter 10, we show that SCP allows a network of nodes to consensuate blockchain contents without forking apart or causing honest nodes to have an incorrect view of those contents. The details of SCP are important, but not necessarily pertinent to users. As far as users need to be concerned, SCP-based blockchains have similar trust requirements to Nakamoto-based blockchains. If a user connects to an honest node,[5] they can be confident they are interacting with the MobileCoin network correctly, but if they connect to a bad node then all bets are off.

9.2.2 MobileCoin network overview

At a high level, the MobileCoin network is not that complicated. Users may run their own nodes to access the blockchain's contents (see Chapter 11 for a more efficient way to use MobileCoin). This allows them to learn about their owned outputs and assemble membership proofs (Chapter 6) for creating new transactions.

Nodes come in two flavors. Validator nodes cooperate with other validator nodes to add blocks to the blockchain by following the SCP protocol. They contain SGX secure enclaves, which are responsible for validating transactions and constructing blocks. Watcher nodes may connect to validator nodes to acquire new blocks, but don't participate in consensuating those blocks.[6] To be clear, validator enclaves only validate transactions and assemble blocks, while SCP is orchestrated by their host processes.

*consensus/enclave/impl/src/lib.rs `tx_is_well_formed()` `form_block()`

To submit a new transaction to the network, a user transmits it securely to the enclave in a validator node. That enclave validates the transaction, then sends it out to the network (i.e. to other validator enclaves). The network follows the SCP protocol, and eventually all validator nodes in the network publish a new block containing the outputs and key images from that transaction (input signatures, membership proofs, and Bulletproofs are discarded by the enclaves, never to be seen by any but their makers' eyes).

[5] In SCP, only honest nodes that are configured well can be trusted. Badly configured nodes may get an incorrect view of the blockchain, although following robust configuration guidelines mitigates that risk [108].

[6] Technically a watcher node could audit published blocks by checking if the key images they contain have appeared before in its copy of the blockchain. Currently watcher nodes only collect blocks from validator nodes and verify the blocks' signatures (via the obscure code path `watcher.rs sync_block() -> reqwest_transactions_fetcher.rs block_from_url() -> archive_block.rs try_from() -> block_signature.rs verify() -> ed25519.rs verify()`), and collect and verify validator enclave AVRs (recall Section 8.2.6) containing block signature keys (which provide evidence that block signatures were created by validator enclaves).

watcher/src/watcher.rs `sync_blocks()` watcher/src/verification_reports_collector.rs `get_verification_report()`

9.2.3 Validation framework

It may seem odd that all transactions can be validated without anyone being able to see them. However, consider a validator node operator's perspective. He was the one to set up the node, so he knows (or can know, if he chooses) exactly what software is being run by the validator enclave. If the enclave software is well made, then no matter what transactions are passed into it, only valid outputs and key images will be appended to the blockchain.

To anchor transaction validation with the existing blockchain state, validator enclaves must verify key images and txout public keys have not appeared in the blockchain, and ring signature members do appear in the blockchain (recall Chapter 7). Both require reaching outside of the `ELRANGE`.

- Key images and txout public keys are sent to the enclave host process, which checks its local copy of the blockchain for duplicates and notifies the enclave of its findings.[7]
- An enclave requests output membership proofs from its host process corresponding to the 'highest indices' from the membership proofs in a transaction-being-validated (recall Section 6.2).

consensus/service/src/validators.rs `is_valid()` `well_formed_check()`

9.2.4 Intentional forking

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

Since the current MobileCoin protocol relies on SGX secure enclaves, periodic hard forks are unavoidable in order to upgrade the network's enclaves with security updates released by Intel.[8] As of writing this, no known hard forks have been executed by MobileCoin validator nodes (the origin block was created on December 7th, 2020 [72]).[9,10]

[7] Validator node operators can use these duplicate checks to connect outputs and key images with specific transactions submitted by users (node operators can see when a user submits a transaction, since she sends data to the validator enclave).

[8] A 'hard fork' results from any change to validator node software that causes updated nodes to create or approve of blocks that would be considered invalid by nodes that weren't updated, or to disapprove of new blocks created by nodes that weren't updated. Validator enclaves can only send transactions to enclaves that share their `MRENCLAVE` value (and have a certain constraint on their `CPUSVN` value to mitigate 'load value injection', see this Intel Security Advisory: [87]), so any enclaves with a non-standard implementation or SGX security version will be unable to access transactions sent to standard enclaves. Such non-standard enclaves will effectively be in a different network from standard enclaves. See Chapter 8 footnote 47 for a discussion of hard forking to enforce SGX security updates.

*crypto/ake/enclave/src/lib.rs `get_verifier()`

[9] While no hard forks have been executed on the main MobileCoin blockchain, the 'testnet' blockchain has successfully undergone forks according to private correspondence with the MobileCoin development team.

[10] Since it is not safe to pass enclave secrets (such as transactions not yet added to a block) to a user-specified new enclave (recall Chapter 8 footnote 28), it is not trivial to transition between validator enclave versions without interruption. One method is to completely shut down the existing network and then reboot it (i.e. all the validator nodes) with new enclave software. Another, more advanced method that avoids interruption, but has not been implemented and is undergoing research, is for the network to automatically transition to new protocol rules by using MobileCoin's consensus protocol (based on SCP) to decide the transition point.

9.3 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 origin message. This is often called an 'airdrop', although increasing regulatory requirements have greatly constrained cryptocurrency developers' ability to distribute coins in this way [152, 153]. Sometimes creators give themselves a large amount in a so-called 'pre-mine' [11].

Second, the currency can be automatically distributed as reward for creating 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 an inflationary cryptocurrency [100]).

9.3.1 MobileCoin money creation

There are currently no well-studied methods for automatically distributing coins in a cryptocurrency employing SCP. As such, all MobileCoins (a.k.a. MOB, MobileCoin's so-called 'stock ticker') were created and distributed in the origin block (see Appendix C).[11,12] There are 250 million MOB in existence, and unless something goes horribly wrong (e.g. a code bug, a catastrophic network failure, or the network hard forking to create more coins), no more MOB will ever be created.[13] Each MOB is divisible into 10^{12} parts called picoMOBs.[14]

[11] According to a filling with the SEC [78], 48 individuals/entities payed approximately 30 million USD to MobileCoin, Inc. under a 'simple agreement for future tokens' (SAFT) in 2018. In a SAFT, a cryptocurrency developer may receive USD from investors prior to the currency's launch, in exchange for documentation saying if the currency is created then the investors will be given 'access' to tokens [73]. Presumably this means some portion of the MobileCoins created in the origin block wound up in the pockets of those 48 people/entities. Since the MobileCoin network is actively processing transactions, it is now impossible (or at least only negligibly possible) to know who on this planet owns how many MobileCoins (without their voluntary disclosure).

[12] It may be possible to layer a proof-of-work-based coin emission scheme on top of an SCP-based cryptocurrency. What follows is a sketch of that system, presented for academic completeness (since MobileCoin has already been launched, there is no possibility of it being implemented there). Add a new 'emission' block type that gets created at infrequent intervals, and two new transaction output types called 'emission claims' and 'emission rewards'. Normal blocks should report fee amounts in the clear (or nodes should keep track of fee totals internally). An emission claim is a simple proof-of-work that takes the last emission block as input, with hash target based on a 'difficulty' value. The claim contains {nonce, `TxOutClaim`} where `TxOutClaim` = {one-time address, txout pub key, `e_fog_hint`}. To build an emission block, we look at each emission claim since the last emission block and create a new emission reward with output amount (communicated in cleartext) equal to a deterministically-calculated block reward, plus the total fee amount that has yet to be consumed divided by the number of emission claims. Each claim gets the same amount. Fees could be pegged to the emission rate in a way analogous to Monero's dynamic fee algorithm [100]. A difficulty adjustment algorithm controls long-term emission rates by making the proof-of-work harder/easier based on how many emission claims have been submitted over time. Note that it is not immediately clear how to implement reliable time-tracking for controlling the interval at which emission blocks are created; perhaps the Stellar Consensus Protocol could be adapted to deciding when to make those blocks.

[13] MobileCoin output amounts are hidden in Pedersen commitments, so the MOB created in the genesis block cannot be observed directly. Instead, observers can generate the 'Origin Account' and examine the outputs owned by that account that originate from the origin block. See Appendix C and [102] for more details.

[14] Incidentally, since Bulletproofs limit output amounts to 64 bits, the maximum amount of MobileCoins that can be contained in one output is approximately 18.4 million MOB (i.e. $2^{64} - 1$ picoMOB), or about $1/13^{\text{th}}$ of the total supply.

9.4 Transaction fees

It would be nice to add any and every transaction to the blockchain. What if someone submits a lot of transactions maliciously? The blockchain, storing outputs and key images from every transaction, would quickly grow enormous. To impose a cost on would-be spammers, MobileCoin transactions are required to include a minimum fee (recall Section 7.2.1).[15]

The fee amounts in a given block's transactions are collected by validator enclaves,[16] which create a fee output that appears the same as a normal output, and publish it as part of the block's content.[17] To ensure all nodes create the same fee output, it is deterministically generated.[18]

*consensus/enclave/impl/src/lib.rs `mint_aggregate_fee()`

- Fee output recipient: When creating a validator enclave, the fee recipient is baked into the enclave software [159]. If any validator node tries to use an 'unofficial' fee recipient, their enclave's `MRENCLAVE` will not match other nodes, effectively shutting them out of the network (validator enclaves can only communicate with enclaves that share an `MRENCLAVE` value).[19]

consensus/enclave/impl/build.rs `main()`

*consensus/enclave/impl/src/lib.rs `form_block()`

- Txout private key r_t: Fee output txout private keys are created within validator enclaves from a hash of the previous block's ID (see Section 9.5) and the current block's full transaction contents. Since part of those contents are discarded before a block is published, observers cannot reproduce the txout private key, leaving the fee output nominally indistinguishable from other outputs.[20]

9.5 Blockchain structure

MobileCoin's blockchain starts with an origin block consisting of 16 outputs that contain the full supply of MobileCoins (see Appendix C). The next block contains a reference to the previous block in the form of block ID, which is a hash of information about the block.

util/generate-sample-ledger/src/lib.rs `bootstrap_ledger()`

[MC-tx] consensus/mobilecoind/src/bootstraps.rs `MINIMUM_FEE`

[15] When MobileCoin was launched [72], the minimum fee was set to 0.01 MOB. It may be possible to replace the minimum fee with, or layer on top of it, an ad hoc fee policy at the network layer for dynamic fee adjustment based on the rate at which transactions are submitted (e.g. to mitigate high-budget attacks [105] and reduce fee costs for normal users by only allowing high fees when transaction volume is increasing [167]), see Chapter 10 footnote 21.

[16] Fees are recorded in picoMOB format in transactions.

[17] Since validator node operators are able to learn which outputs come from which transactions submitted by users (recall Section 9.2.3), they can also figure out which outputs published in each block are fee outputs. It is impossible to control the behavior of validator node operators, so it's reasonable to say the classification between fee outputs and normal outputs is 'public knowledge', although in practice most fee outputs could go unidentified. It may seem like eliminating duplicate checks for txout public keys could prevent operators from identifying fee outputs, however operators can set up a node running on a divergent network, which receives transactions from the normal network, but publishes blocks independently. If this divergent network has different block IDs, then fee outputs will be different from normal-network fee outputs, allowing the node operator to discern which real-chain outputs are for fees.

[18] Fee amounts (and, as it happens, the tombstone blocks) from each transaction are transmitted out of validator enclaves and explicitly readable by node operators. Supposedly this allows node operators to inform users about fees in recent blocks, so they can make transactions with higher fees if the network is congested.

consensus/enclave/api/src/lib.rs *struct* `WellFormedTxContext`

[19] In the launch version of MobileCoin, fees are supposedly received by the MobileCoin Foundation. [102]

[20] If a specific transaction author created all the transactions in a given block, then he will be able to recreate the fee output's txout private key. However, this provides him no benefits as he can already deduce which output in the block wasn't created by him (and hence must be the fee output) and the global fee recipient is public knowledge.

As mentioned, new blocks are created based on the collaboration of many nodes in the network (see Chapter 10). Once the network decides to add a block of transactions to the chain, the network's validator enclaves discard extraneous transaction information (ring signatures, membership proofs, and Bulletproofs) and publish the remaining parts (key images and outputs).[21]

Importantly, validator enclaves that publish a block they participated in validating also sign the block.[22] The signing key is the same key obtained by transaction authors when setting up a communication channel with validator enclaves, meaning it has a corresponding `AVR` (recall Section 8.2.6). This allows observers to verify that blocks recorded in the blockchain were validated by SGX secure enclaves running the MobileCoin validator node software.[23]

*consensus/enclave/impl/src/lib.rs `form_block()`

9.5.1 Blocks

A block is a block header, a collection of key images and transaction outputs, and a block signature.[24] Block headers record information about each block. We present here the outline of a block's contents. Our readers can find a real block example in Appendix B.

- Block header:
  - **Block ID**: This block's ID, created as a hash of the other block header fields.
  - **Version**: This block's version (0 at the time of MobileCoin's launch). Semantically, the block version can be thought of as the number of scheduled hard forks (i.e. protocol updates) that have occurred since MobileCoin launched.[25]
  - **Parent block's ID**: Referencing the previous block, this is the essential requirement of a blockchain.
  - **Index**: This block's index in the blockchain (i.e. among all the blocks that exist).[26]

[MC-tx] src/blockchain/block.rs `compute_block_id()`

[21] Since the network actively collaborates on new blocks, new blocks only appear when there are transactions to process. That is to say, the blockchain will not be modified whenever there are no transactions being submitted.

[22] Blocks are signed with a Schnorr-like signature scheme using curve Ed25519 (without the Ristretto abstraction) implemented in the `dalek-cryptography ed25519-dalek` library [44].

[dalekEd] src/secret.rs `sign()`

[23] Due to an oversight in MobileCoin's launch process, watcher nodes were not equipped to collect validator enclave `AVR`s until two months after the origin block was created [60], while validator nodes themselves only retained copies of `AVR`s for active signing keys. Moreover, in the period between launch and watcher nodes beginning to collect `AVR`s, all active validator nodes restarted their enclaves at least once. The practical effect of this is there are no records of `AVR`s from the first 479 blocks (created during the first three days of MobileCoin's existence). However, those 479 blocks have at least six signatures each, supposedly corresponding to the six nodes active at the time of MobileCoin's launch (this has not been verified yet as far as we know). This means, at the very least, that for those initial 479 blocks to contain forgeries that minted new coins, all six node operators must be dishonest (they either lied about their signature claims, or collaborated on a block forgery). If `AVR`s were available, then for forgeries to occur, all six node operators must have both collaborated and exploited an SGX vulnerability simultaneously. In other words, the 'trust model' for assessing the integrity of those blocks is reduced from 'node operators and Intel' to just 'node operators'.

[24] Before a block is published (and before its block ID is calculated), its key images and outputs are sorted (outputs are sorted by txout public key). This ensures all validator enclaves end up creating identical blocks.

*consensus/enclave/impl/src/lib.rs `form_block()`

[25] Future versions of MobileCoin may update the block header to reflect the hard fork-based versioning scheme more precisely [169].

[26] The origin block has index '0'.

  - **Cumulative output count**: The total number of outputs in the blockchain at the time this block was created, including this block's outputs.
  - **Root element**: The root hash of the outputs currently recorded in the blockchain (excluding this block's outputs). In other words, the root hash of any membership proof created from the existing output set.
  - **Contents hash**: A simple hash of all the key images and outputs appended to the blockchain by this block (calculated after they are sorted).

*consensus/enclave/impl/src/lib.rs form_block()

- Block contents: All the key images and transaction outputs (including the fee output) to be appended to the blockchain by this block.

[MC-tx] src/blockchain/block_contents.rs hash()

- Block signature: A signature on the block header by a validator enclave. Specifically, each validator enclave that publishes a block (many enclaves will publish each block concurrently) creates their own signature and includes it with their local copy of the block. By collecting the block signatures and `AVR`s from many validator nodes in the network, users can gain confidence that blocks are legitimate according to the MobileCoin protocol rules.[27]

[MC-tx] src/blockchain/block_signature.rs form_block_and_key-pair()

To record a new block, the following information is stored:

- Block header: Aside from the 4-byte block version, 8-byte block index, and 8-byte cumulative output count, all parts of the block header are 32 bytes. In total, a block header is 148 bytes.
- Block contents: Recalling Section 7.3, each transaction output is 188 bytes and each key image is 32 bytes.
- Block signature: A block signature contains a 64-byte signature, 32-byte public signing key, and 8-byte timestamp added by the node operator. In total, 104 bytes.[28]

[dalekEd] src/constants.rs SIGNATURE_LENGTH

9.6 Beyond the validation framework

MobileCoin is the first cryptocurrency to try and completely erase parts of its transaction history. In all other cryptocurrencies, even if some block contents are automatically pruned by nodes (e.g. Mimblewimble/Grin [38]), in all cases full versions of transactions can be copied and stored indefinitely by node operators.

This poses an interesting problem for future users of MobileCoin, who cannot re-validate existing transactions. How can they be sure the rules discussed in Section 9.1.1 have always been applied correctly?

[27] Once a block has been sent out of a validator enclave, the host process adds a timestamp next to the block signature. This timestamp is not verifiable in any way, but helps observers of the blockchain get an idea about when blocks were created.

consensus/service/src/tx_manager.rs tx_hashes_to_block()

[28] A simple block representing one transaction with two inputs and two outputs will be $148+2*32+3*188+104 = 880$ bytes (recall there is a fee output added to each block).

While transactions cannot be re-validated, there remain several pieces of information that observers can use to assess the MobileCoin blockchain. Above all else, it should be the role of watcher nodes and watcher node operators to collect, actively examine, and archive these data.[29,30]

1. The implementation of validator nodes and validator enclaves. Quite simply, if transaction validation and block construction are not implemented properly, then MobileCoin can't possibly work. MobileCoin's source code is presently stored in a Github repository [71], and can be examined by any person on the planet (who can access Github). Github may become unavailable in the future, so observers should maintain copies of the repository for future reference.

   Note that 'git' repositories inherently store a complete history of changes, so it's reasonably easy to examine old versions of the MobileCoin implementation given a copy of the repository.

2. To connect the supposed implementation of MobileCoin validator nodes with the actual validation code executed at any point in time, blocks are signed with keys that are attested to in `AVR`s. These `AVR`s have `MRENCLAVE` and `CPUSVN` values representing the code that validator enclaves are running and which SGX implementation they are using (recall Section 8.2.6).

   A future observer can use the block signatures found in the blockchain, in combination with their signing keys' `AVR`s (which can be collected by observers from validator nodes), to gain confidence that blocks were validated properly. However, SGX secure enclaves are prone to vulnerabilities [8], so a given `AVR` may not be reliable. As more `AVR`s and block signatures are collected for each block in the blockchain, it becomes less likely any of the blocks are invalid. So long as just one validator enclave represented by a known valid block signature `AVR` was operating correctly while creating a block, the block must be correct.

3. Enclaves are only able to validate blocks with respect to information made available by the local machine. Specifically, key image checks (and txout public key checks) are performed by the local machine, and current membership proofs are provided by the local machine to the enclave (recall Section 9.2.3).

   While a validator node operator could fool their enclave into approving a transaction containing a key image that already exists in the chain, this is easy to detect. Anyone who obtains a copy of an invalid block created by that node will see the duplicate key image and can raise an alarm (likewise with a duplicate txout public key).

   More insidiously, a validator node operator could falsify the record of existing on-chain outputs, and provide membership proofs to their enclave representing outputs that don't actually exist in the chain. This could allow a conspiring transaction author to create MOB out of thin air by spending those fake outputs.

[29] Watcher nodes may be operated by anyone in the world, and it is our view that no amount of watcher nodes running concurrently is too many.

[30] As mentioned, the current watcher node implementation only collects blocks and `AVR`s, and verifies that block signatures are correct. Hopefully future updates to that implementation will expand its capabilities.

To avoid that problem, each block header includes a root element (recall Section 9.5.1), which is the root hash of any membership proof made with the current set of on-chain outputs. The MobileCoin protocol requires that all highest-index membership proofs passed to an enclave for validating the transactions in a block have the same root hash (the same one which is placed in the block header) (recall Section 6.2). Observers of the blockchain can recompute the root element for each block, and ensure it correctly represents the full set of outputs added to the chain before that block.

4. Even if a specific block appears valid to observers, the enclave that created it may be interacting with a partially invalid or malformed blockchain. This can manifest in different block IDs between nodes. To promote uniformity in the network, it is important to check the correctness of block IDs as blocks are produced. In particular, watcher nodes should only accept information about blocks with apparently valid block IDs (in other words, there is only one valid blockchain).

5. While very rare under normal conditions, it is possible for the network to fork into more than one fully valid chain (called a 'soft fork'). These soft forks must be resolved by part of the network deliberately abandoning their chain and adopting the other part's chain (see Section 10.4.2).[31]

   In the Nakamoto consensus protocol, small 'fully-valid' forks are expected and common, but in MobileCoin's consensus protocol they should be rare and unexpected (indicating a serious problem with the network). As such, each spontaneous fork that comes into existence and gets resolved must be recognized explicitly by node operators in the network. Those operators should seek to understand why there was a fork, and work to prevent future forks from happening.

   Observers, especially watcher nodes, should record spontaneous forks so they can keep validator node operators accountable for network health, and log for future reference these critical events.

In SCP, it is necessary that there be some honest node operators [115], but MobileCoin's implementation loosens this restriction by tying the validity of block contents to secure enclaves. As we have described, even if all node operators are dishonest, unless they all coordinate a specific attack that relies on an SGX vulnerability, observers (e.g. watcher nodes) have the ability to detect suspicious behavior (e.g. coin-minting or double-spends).

[31] Malicious node operators may try to exploit the soft fork resolution design to execute a double spend attack. In other words, they could spend outputs on one side of the fork, then convince the network to adopt a different side of the fork where those outputs weren't spent. In practice, this would be very difficult to accomplish without compromising large parts of the network (see Chapter 10).

CHAPTER 10

MobileCoin Consensus Protocol

Federated Byzantine agreement (FBA) is a model for reaching consensus in a network composed of nodes. Its most essential feature is the quorum slice, from which quorums and the federated voting procedure are derived. A consensus protocol is said to employ the FBA model if it relies on quorum slices and federated voting. The Stellar Consensus Protocol (SCP) is one example, wherein nodes reach consensus on abstract 'composite statements'. MobileCoin's consensus protocol is a concrete implementation of SCP where the composite statements consensuated are blocks to be added to the blockchain.

The material in this chapter was primarily sourced from Mazières' original SCP whitepaper [115] and MobileCoin's source code [71], with many of the supporting arguments based on original research. Additional useful resources include [117], [77], [108], [116], [24], and [109].

10.1 Quorum slices and quorums

Let's consider the perspective of a single node in a distributed network of many nodes. This node participates in the network to learn about (and help create) arbitrary 'statements' that other nodes agree to accept as permanent facts. For example, the node may want to validate and maintain a useful local copy of a cryptocurrency blockchain. A 'useful' blockchain is one that accurately records all transactions that users of the cryptocurrency are likely to submit future transactions with respect to.[1]

[1] In the case of MobileCoin, a useful blockchain records all transaction *outputs* (and key images) that future transactions will be submitted with respect to.

However, it is not obvious which other visible nodes in the network are maintaining a useful blockchain. Any node could be malicious or inaccessible to users.[2] As a consequence, our node must choose to *trust* some group of nodes. Practically speaking, if a trusted group of nodes all agree with our node on what block should be added to the blockchain next, then our node should be willing to accept that result.[3] We call trusted groups *quorum slices*.

It makes sense for all nodes in the network to operate using the same rules. If a node only accepts results that its quorum slice agrees on, but the nodes in that slice also require agreement from their own slices, then a result can only appear when agreement is reached between all the unique nodes found by tracing slices. The final group of nodes (the 'transitive closure' of quorum slices) is called a *quorum*.[4] Keep in mind that a quorum always includes the node that defined it.

consensus/scp/src/quorum_set.rs `findQuorumHelper()`

Quorums are defined from the point of view of a single node's quorum slice, so it is likely that different nodes will see different quorums. However, if two nodes' quorums overlap, then it isn't possible for the two nodes to finalize different results (e.g. decide to add different blocks to the blockchain), because it isn't possible for the overlapping nodes in their quorums to accept more than one final result. It is this property of *intersecting quorums* that allows networks to reach broad consensus on results.

10.1.1 Multiple quorum slices

To complicate matters a bit, nodes are free to define multiple quorum slices. However, each node only contributes one quorum slice to each quorum it is a member of. This means if nodes in the network each have many quorum slices, then each node will see a large number of quorums corresponding to the various combinations of slices it can trace out.

Defining more than one slice is useful in case a node in one slice gets compromised by an attacker or is unresponsive. Such misbehaving nodes could block the slice's owner from seeing agreement for a full quorum, or convince the owner it should accept a result inconsistent with a different part of the network.[5,6]

[2] In a blockchain-based technology using proof-of-work, nodes can find the most useful blockchain by hunting for the chain with the highest cumulative difficulty [130]. Cumulative difficulty reflects energy expenditure, so it can't be counterfeited without breaking the proof-of-work hash algorithm. This makes it a reliable signal for identifying consensus.

[3] This dynamic where nodes willfully trust other nodes is similar to the relationship between cryptocurrency users and nodes. Users have no choice but to trust the nodes they communicate with.

[4] A 'quorum' is generally considered the minimum number of participants in a deliberative system that must agree in order for the system to make a decision on the content of deliberation.

[5] Misbehaving nodes are said to represent 'Byzantine failures', after the famous Byzantine generals problem [106]. FBA includes the word 'Byzantine' because it is designed to allow network agreement even in the presence of (some) Byzantine failures.

[6] There is a trade-off to consider when defining quorum slices. Small slices lead to small sets of overlapping nodes between intersecting quorums, which means if there are enough misbehaving overlapping nodes, they could break quorum intersections by telling different nodes they want to finalize different statements. The end result could be network divergence, where parts of the network accept different results. Large sets reduce that risk by increasing the number of nodes in quorum intersections, so more nodes must be compromised to convince a quorum to accept a divergent result. On the other hand, a misbehaving node could be unresponsive, blocking any quorum it participates

Even with many quorum slices, if enough slice nodes are misbehaving, then a well-behaved node can still get blocked or poisoned into accepting results inconsistent with other parts of the network (which are healthy). Such a node is called *befouled*, while healthy nodes are *intact*. Ultimately a befouled node has trusted the wrong nodes, and must take responsibility for those bad decisions. A consensus protocol based on quorum slices can only guarantee broad consensus on results for intact nodes.

10.1.2 Quorum sets

Exactly how node operators define their quorum slices is an implementation detail which we would normally neglect, however the approach used by default in MobileCoin will be important to understand for Section 10.3.1.[7]

It may be the case that a node operator trusts some nodes more than others. Well-trusted nodes can be members of relatively more quorum slices, because they are less likely to misbehave (according to the node operator's estimation). To that end, quorum slices are generated automatically from a tree-like *quorum set* structure.

consensus/scp/src/quorum_set.rs *struct* `QuorumSet`

The node's quorum set starts with a 'parent set'. This parent set is composed of nodes and child sets, which are both called *members*. Each child set is in turn composed of nodes and grandchild sets, and so on. Every set has a *threshold* defined by the node operator, which is the number of members that set can contribute to a given quorum slice.

To define an arbitrary quorum slice, take random members from the parent set until its threshold is met. If any of those members is a child set, take members from that child set until *its* threshold is met, and so on.

When a set's threshold is close to the total number of members in the set, each of those members will participate in a relatively higher share of the quorum slices of which the set is a member, compared to if the ratio was lower. Moreover, members close to the parent set (or in the parent set) will participate in relatively more quorum slices than members farther away. Node operators can use those two variables to place well-trusted nodes in more quorum slices than less-trusted nodes.

10.2 Federated voting

Suppose nodes in the network wish to reach agreement on an arbitrary question. In the end, they could agree on multiple answers to the question, or just one answer. We call a potential answer

in from finalizing any result, so the more quorums it is a member of, the more it can block. It is considered better for agreement to fail than for the network to diverge, because pursuit of agreement can be iteratively attempted, while healing divergences is a risky endeavor since finalized results may have irreversible effects on the network's users. For this reason, large quorum slices are preferable.

[7] While implementation details are mostly out-of-scope for this document, they are not unimportant. As discussed in [108], it is imperative that nodes configure their quorum slices based on robust guidelines, lest the network be prone to diverging (disconnected quorums) or halting (no full quorums are available due to outages).

a *statement*. Statements can 'contradict' each other, which means they are not allowed to coexist in the final set of answers. Whether statement S is contradicted by statement S', or whether S or S' even qualify as valid answers, are rules defined by whoever wrote the question and enforced by each node individually.

10.2.1 Level 1: voting

Any node can offer a statement S to the network by issuing a level 1 declaration (a.k.a. a *vote*) for S, which it transmits to other nodes. Those nodes can then issue their own votes for S, so long as S is valid and they have not already issued a vote for a statement that contradicts S (or accepted one).

There are two basic classes of statements. So-called *irrefutable* statements can't be contradicted by any other statement, according to whoever defined the question being answered. Once a node sees a full quorum of votes for an irrefutable statement S (including itself), then S has full agreement and can be finalized immediately. It is only a matter of time before other nodes in the network learn about S, see a full quorum, and finalize it.

Naturally, the second class is statements that can be contradicted. Unfortunately it isn't enough to see a full quorum of votes for a statement S that is contradictable. What about nodes outside that quorum? Any of them could have voted for a statement contradicting S, or could do so in the future. All quorums must intersect with each other for a healthy network, so if a full quorum has voted for S, then no quorum can vote for an S-contradicting statement S'. Without agreement between intersecting quorums, the network is likely to have many broken nodes unable to finalize any statements.

10.2.2 Level 2: accepting

Instead of finalizing a contradictable statement S voted for by a quorum, a node instead *accepts* that S has what it takes to be a final answer. If a node v accepts S, it will issue a level 2 declaration to other nodes informing them it accepts S. A level 2 declaration also implies a level 1 declaration (vote) for the same statement.

Whenever a quorum seen by v has voted for S (or accepted it), then so long as all other nodes in the quorum are well-behaved, those nodes will also eventually accept S. This is because all those nodes also have a quorum of agreement on S, even if their quorums are only subsets of v's quorum.[8]

Meanwhile, once a solid quorum of votes exists, nodes in quorums that intersect with the solid quorum will be unable to accept any statements that contradict S, because the overlapping nodes will have already voted for S. Of course, there could be quorums that don't intersect with the

[8] If node $v2$ sees a quorum Q2, and $v2$ is a member of v's quorum slice QS, then all the nodes in Q2 will be members of quorum Q made from QS (assuming all nodes only have one quorum slice each, for simplicity).

solid quorum, which could have their own solid vote for a contradictory statement. However, disconnected quorums should not exist in a healthy network and are the responsibility of node operators to prevent by selecting good quorum slices.

When a node x has accepted statement S, they are ready to finalize it but are waiting for other nodes to get on board (more on that in a bit). If a different node v has only voted for statement S' (which contradicts S), but hasn't accepted it, then it is reasonable to think v could be convinced to change his mind.

Suppose at least one node in each of v's quorum slices has accepted S, and is waiting to finalize it. Clearly there is no way v will ever see a full quorum of votes for S', as all his possible quorums are blocked by at least one level 2 declaration for S. We call those nodes a *v-blocking set*, because they block v's prior vote.[9,10] Since v's vote for S' is hopeless, it is safe for him to abandon that vote and accept S, which has a higher chance of succeeding.[11] More generally, whenever a node sees a v-blocking set for statement S and it hasn't already accepted a statement contradicting S, it will automatically accept S, and abandon any votes for contradictory statements.[12]

It's worth emphasizing that a v-blocking set is composed of one node from each of v's quorum slices, rather than at least one node from each of v's quorums.[13] Intuitively, this is because v only trusts his quorum slices, and shouldn't be willing to abandon a vote based on the claims of nodes he doesn't directly trust.[14]

To reiterate, once a statement has been accepted (in a well-connected, well-behaved network), no other statement can be accepted by any node. Imagine a quorum Q that has voted for statement S, allowing a member of the quorum v to accept it. For a node n outside of Q to accept a statement S' that contradicts S, it either must see a full quorum voting for S', or see a v-blocking set accepting S'. Since a v-blocking set can only appear when at least one quorum has accepted S', the problem

[9] If there is a v-blocking set of befouled nodes (misbehaving nodes or nodes blocked/confused by misbehaving nodes), then v is blocked from seeing a full quorum of agreement of honest nodes. In that case, v would also be considered befouled. [115]

[10] In MobileCoin, quorum slices are defined in terms of quorum sets (Section 10.1.2). A quorum set contains a v-blocking set if, in the parent set, there are so many blocking nodes that non-blocking members can't meet the parent set threshold. This is why we use the term 'threshold' — it's the minimum number of members that need to agree with you for your node to not be blocked. This idea extends to members of the parent set. Any child set with enough blocking members to prevent the set from reaching its threshold qualifies as a blocking member of the parent set. consensus/scp/src/quorum_set.rs `findBlockingSetHelper()`

[11] Keep in mind that nodes are agnostic about which statements get finalized. Their only goal is to remain consistent with at least one of their quorum slices, by either reaching final agreement on *some* valid statements, or not finalizing any statements if necessary.

[12] If v-blocking sets were based on votes instead of level 2 declarations, the network would be unstable. Imagine a node $v2$, where v-blocking sets based on votes are allowed. Node $v2$ is a member of a set that blocks v, which convinces v to vote for S. It is possible that $v2$ at a later date sees a $v2$-blocking set that causes it to change its vote from S to S"! Malicious nodes could trap the network in cycles of vote flipping.

[13] If a single node is present in multiple quorum slices, then it can 'block' all of those slices. In other words, a v-blocking set doesn't have to be composed of only unique nodes from each slice.

[14] More technically, if v were blocked by a node in each of his quorums, then if those blocking nodes weren't in his slices, his slices would be transitively blocked since the blocking nodes would be sourced from his slice's slices. Checking for v-blocking sets among local quorum slices is a safe simplification, because given enough time, a v-blocking set based on quorums will reduce to a v-blocking set in the local quorum slices as those slice nodes realize they are blocked.

reduces to creating a quorum that votes for S'. However, a quorum has already voted for S, so since all quorums intersect, there are no full quorums available to vote for S'. Therefore S' cannot get accepted.

10.2.3 Level 3: confirming

Now imagine a quorum Q whose nodes have all accepted statement S. Even though no other statement has a hope of being accepted, suppose anyway that all nodes N' outside of Q have voted for S'. Are they doomed to remain in disagreement with Q? Oddly, we claim at least one node v in N' has a v-blocking set whose nodes are all in Q.

To justify that claim, first recall that quorums are created by tracing out quorum slices, so if nodes have many slices, there can be many different quorums. Imagine every node in N' has at least one quorum slice that doesn't have a node in Q. In that case, it is guaranteed there is at least one quorum Q', seen by at least one node, created by tracing some of those slices, which does not intersect with Q. However we said the network is well-connected, so Q' is not allowed, since otherwise the nodes in Q' could finalize a result different from the nodes in Q!

On the other hand, if there are nodes V in N' which each have a v-blocking set in Q (even if V just contains one node), then so long as all the other paths that can be traced by quorum slices from nodes in N' end up hitting a node in V or Q, all quorums made by nodes in N' will intersect with Q. Note that it isn't necessary for any of the nodes in N' - V (nodes remaining if nodes from V are removed from N') to have quorum slices with nodes in Q if they are able to access Q by linking to a node in V.

The essential consequence of this v-blocking set idea is once quorum Q has fully accepted statement S, there will always be nodes V (in a well-connected network) which it can persuade to also accept S. Moreover, we can extend the previous analysis to the nodes in Q + V. If there are no nodes in N' - V with a v-blocking set in Q + V, then there must be at least one node in N' - V that sees a quorum that doesn't intersect with Q, which is not allowed. Therefore in a well-connected and behaved network, once a quorum Q has fully accepted a statement S, all other well-connected and behaved nodes will eventually also accept S, as v-blocking sets gradually eat away at N' (there is a 'cascade' effect).

Since a full quorum Q accepting S means the entire network will eventually accept S, we have found an adequate and robust condition for finalizing statements. Once a node sees a full quorum of nodes accepting statement S (including itself), it will *confirm* S, and record S for permanent use. Eventually the entire network will also confirm S.

It's possible for many statements to be voted on and ultimately confirmed in parallel, so long as they don't contradict each other. Likewise, it's possible the network will never get to the point of confirming any statement, if too many nodes vote for contradictory statements. The Stellar Consensus Protocol discussed in Section 10.3 is designed to address that problem by defining a set of rules and heuristics that enable 'eventual' consensus despite any contradictions.

10.2.4 Revisiting befouled nodes

So far we have focused on well-connected and -behaved networks. We showed that nodes in such networks only finalize statements that will also be finalized by all other well-connected and -behaved nodes. At what point can misbehaving nodes prevent honest nodes from having that finalization guarantee?

Clearly each misbehaving node can poison any quorum of which it is a member, preventing it from reaching agreement on any statement. In other words, there must be at least one fully well-behaved quorum for the network to finalize any statement.[15]

Any node v with a v-blocking set of misbehaving nodes is at risk of never seeing a full quorum of agreement on any statement. Moreover, the v-blocking set could convince v to accept a statement S' which contradicts other statements accepted by nodes in the network, causing v to appear like it is misbehaving to other nodes. As we have noted before, 'befouled' nodes are those that are both inherently misbehaving, and those that have been poisoned by other befouled nodes.

If a befouled node provides quorum intersection to other nodes, it can disconnect those nodes from parts of the network. It is helpful to think about network health in terms of the set of well-behaved nodes that remains when all befouled nodes have been removed (i.e. ignored). If that set is well-connected, then there is no risk of quorums finalizing different statements. Of course, that guarantee does not exist for befouled nodes (even if they are honest), who may fail to finalize a statement finalized by the rest of the network, or even finalize a contradictory statement.[16]

10.3 Stellar Consensus Protocol

The Stellar Consensus Protocol (SCP) is an abstract implementation of the federated Byzantine agreement (FBA) model discussed in the previous sections. The protocol is concerned with answering one high-level question, by asking a series of intermediate questions subject to federated voting (with various additional rules). The answer to that high-level question is a *slot*, which can be *externalized* for permanent use by nodes (slots are the outputs from the protocol).

Rather than specify what high-level question must be answered, SCP leaves it open ended. MobileCoin's consensus protocol is a concrete implementation of SCP, where the question is "what should go in the next block of the blockchain?". Its answer is the set of transactions that will go in that block, while the block is considered a slot.

This section is mainly concerned with elucidating the Stellar Consensus Protocol. Occasional notes about MobileCoin-specific implementation details are added where appropriate.

[15] If enough misbehaving/befouled nodes agree with each other, then they could form a quorum that finalizes different statements from the rest of the network. However, misbehaving nodes can do whatever they want, including finalizing any random statement at any time, so a misbehaving divergent quorum is not considered a problem any protocol can address.

[16] See `https://stellarbeat.io/` for an example of how an SCP-based network's health can be evaluated and monitored, in terms of quorum intersections, and section 6 of [108] for a discussion about how to evaluate network health efficiently.

10.3.1 Nomination

After spending so much of Section 10.2 discussing contradictable statements, it may seem like we intend to make the statements "Add Tx1 to Block X" and "Add Tx2 to Block X" contradictory if Tx1 and Tx2 try to spend the same outputs (have overlapping key images). The problem with this is it allows Tx1 and Tx2 to become stuck if no complete quorum votes on just one of them, which may prevent the network from finalizing Block X (e.g. if Tx1 and Tx2 are the only transactions available). A more efficient approach, which is also observed in Nakamoto consensus, is to finalize one of them and discard the other.

In the first phase of SCP, called *nomination*, nodes may nominate apparently contradictory values for the next slot. For example, it is valid for a node to vote for both of the statements "I nominate Tx1 for Block X" and "I nominate Tx2 for Block X".[17] The network performs federated voting on these nomination statements.

consensus/scp/src/slot.rs `do_nominate_phase()`

If nodes can vote for new nominations without limit, then the nomination stage could last indefinitely, with new statements constantly being passed through. However, the goal of SCP is to reach consensus on a set of nominations and finalize a slot before moving to the next slot. It is fine for statements that don't make it into one slot to go in the next one.

To that end, once a node confirms any nomination, it will no longer vote for new nominations. In other words, confirmed statements always contradict statements that 'could be voted on', preventing votes from being cast. However, v-blocking sets can always convince a node to accept a given nomination. This loophole is necessary because any node that has accepted a statement is likely to confirm it at some point (see a full quorum of acceptance), and we want the network to reach consensus on confirmed nominations. Ultimately, all statements that have progressed a fair distance into the nomination stage will be confirmed (typically any statement that gets a full quorum of votes), while others will get stuck and may be re-nominated in future slots. Importantly, at least one nomination will always be confirmed by the entire network.

While clear that if enough time passes the network will converge on a set of confirmed nominations, it isn't possible for nodes to directly know when that happens. However, convergence can be discerned indirectly by performing iterative federated voting on nomination sets. If a quorum reaches agreement on some nomination set, then the network has converged and that set can be finalized for the SCP slot under deliberation. We go into more detail on this *balloting* procedure in Section 10.3.2, but for now must take two small detours.

Combining nominations

Nominations containing values that are apparently contradictory can be confirmed together. Naturally, those values can't coexist in the final slot. As such, implementers of SCP must define a

consensus/service/src/validators.rs `combine()`

[17] In MobileCoin, transactions are only visible to the code executed by secure enclaves, while SCP is implemented outside of enclaves. Practically speaking, a nominated transaction is really a nominated transaction ID (a hash of the transaction), and SCP nodes just pass those IDs around. Connecting transaction IDs with actual transactions, and validating those transactions, is implemented as supporting functionality alongside the main SCP protocol.

[MC-tx] src/tx.rs `tx_hash()`

method for combining nominations into a *composite value*. Each composite value is a candidate for the final slot value, and only one composite value can win in the end.

In MobileCoin, nominated transactions are combined by sorting the nomination set, then removing transactions with at least one key image or txout public key that already exists in the list (i.e. in a transaction with lower index).[18] Transactions are sorted by fee (highest fees first), then by transaction ID in lexicographic order ('smaller' IDs first).[19,20] A composite value in MobileCoin is therefore a list of transactions that can be directly used to form a block.[21]

consensus/enclave/api/src/lib.rs `cmp()`

Federated leader selection for voting

In SCP, nodes do not freely vote for every nomination they see from other nodes. Instead, they select 'leader' nodes and only vote for the nominations voted for by those leaders. Supposedly this reduces the total amount of data passed between nodes in the network by limiting who will cast votes and when [115]. However, we are not aware of any research backing this up, and are not convinced it has any practical effect on total data passed between nodes compared to free voting. Free voting would seem to require relatively fewer communication rounds between nodes, lessening the data transmitted in that approach. In any case, since leader selection is actively used in MobileCoin, we will describe it here.

The nomination phase of SCP is implemented as a series of 'rounds'. Each node executes the same series of steps in a given round before starting a new one (different nodes in the network don't have to be synchronized; rounds are only a 'local' concept). During a round, a node collects messages from other nodes which it will use in the next round. Each round, the node adds a new leader to its list of leader nodes. After adding a new leader, it checks if any messages received from any of those leader nodes contain nomination votes or level 2 declarations (acceptances) which it hasn't

consensus/scp/src/slot.rs `do_nominate_phase()`

[18] A transaction removed from a nomination set is not immediately considered invalid by the node. It will either be re-nominated for the next block if not invalidated by the current block (i.e. if the final version doesn't contain a key image or txout public key in the removed transaction), or could even be finalized for the current block if a different node retains it in their own composite value and that value wins out in the balloting procedure discussed in Section 10.3.2.

[19] A transaction's ID is a hash of all its contents.

[MC-tx] src/tx.rs `tx_hash()`

[20] Combining nominated transactions is implemented outside of validator enclaves, because all of the combination rules use transaction information that is known to validator node operators. When we say 'transactions' are combined, in practice it is truncated versions of the full transactions that are combined. The full versions are referenced by transaction IDs.

consensus/enclave/api/src/lib.rs *`struct`* `WellFormedTxContext`

[21] In the version of SCP that has been implemented up to the time of writing this, nodes are assumed to have a black-and-white view of statements that could be nominated. Either a statement is valid, in which case it will always be voted for by all nodes, or it is invalid and no node will vote for it. However, the nomination phase is effectively a procedure for the network to hold referenda on potential statements. This means it is possible (and safe) for nodes to express opinions about those statements. If a node thinks a statement is valid but undesirable, then they are free to not vote for it. If an undesirable statement gets approved by enough nodes, then nodes that did not vote for it will have to accept it due to v-blocking sets (we call this 'tipping-point' voting). One trivial application of discretionary voting is implementation of an ad hoc fee policy at the network level. Only if enough nodes believe a transaction's fee is 'high enough' will the transaction gather enough votes to be accepted nominated, and eventually confirmed nominated. Another more advanced application would be referenda on system-wide parameter changes like version numbers [169]. Note that a malicious set of nodes large enough to v-block at least one quorum can convince the network to confirm any valid statement, so deliberative voting is only useful if there are sufficient honest nodes.

voted for before (or accepted). Assuming the node hasn't confirmed any nomination yet, it issues new votes for those not-voted-for nominations.

A node selects leaders from among its quorum slices. Only when a node selects itself to be a leader will it issue brand new votes for statements it wants to nominate. Leaders are selected with the following algorithm.

1. Node v computes a *weight* for each node in its quorum set, based on the fraction of quorum slices it participates in. A member of the parent set participates in

   $$(\frac{parent\_threshold}{num\_parent\_members})$$

   quorum slices. If that member is a child set, then its own members participate in

   $$(\frac{child\_threshold}{num\_child\_members}) * (\frac{parent\_threshold}{num\_parent\_members})$$

   slices, and so on.

   Only the first appearance of a node in v's quorum set is used to compute its fraction (even though it may be a member of multiple child sets), to reduce the algorithm's complexity. Since v participates in all of its own quorums, its fraction is automatically 1.

   A node's weight is its fraction times the maximum number that can be stored in a 32-byte variable (i.e. $2^{256} - 1$).

   consensus/scp/src/quorum_set.rs `weight()`

2. Decide which nodes in the quorum set are eligible to be leaders this round by computing a 32-byte hash for each node and comparing it to the node's weight. Only if the weight is higher than the hash output is the node eligible. The hash contents are: slot index (i.e. block index), 1 (an integer for differentiating the hash result from the priority computation in the next step), round number, and node ID (a public key associated with the node).

   $$hash = \mathcal{H}_{32}(slot\_index, 1, round\_num, node\_id)$$

   Less trusted nodes will be eligible to be leaders less often than more trusted nodes (recall Section 10.1.2), since their weights will be lower.

   consensus/scp/src/slot.rs `neighbors()`

3. Compute a *priority* for all the eligible nodes.

   $$priority = \mathcal{H}_{32}(slot\_index, 2, round\_num, node\_id)$$

4. Add the node with the highest priority to the list of leader nodes. If the highest priority node is already in the list, use the next highest priority node, and so on.

   consensus/scp/src/slot.rs `find_max_priority_peer()`

If enough nomination rounds pass, then all nodes in v's quorum set will be considered leaders, so federated leader selection does not affect how SCP works at a fundamental level.[22]

[22] It might seem that federated leader selection for voting would make propagating transactions throughout the network slow and inefficient. However, transactions are transmitted between nodes (i.e. node enclaves) independent of SCP, which only deals with transaction IDs. Nodes transmit transactions to each other with a simple 'flood' protocol, in which newly encountered transactions are sent aggressively to all known nodes. Unfortunately flooding transactions allows network observers to estimate which node a given transaction was originally submitted to. A more sophisticated technique called Dandelion++ involves first sending new transactions through a line (or 'stem') of nodes, before flooding ('fluffing') the transaction to the network [62].

consensus/service/src/consensus_service.rs `create_scp_client_value_sender_fn()`

10.3.2 Balloting

Once a node has a composite value created by the nomination stage, it is faced with two conflicting motivations. On the one hand, it wants to find a quorum of nodes with the same composite value so the value can get accepted, confirmed, and ultimately used as a permanent output of the protocol. On the other hand, if there are no quorums available for the current composite value, it wants to keep updating that value until a quorum does appear (one should appear eventually because nomination is designed to converge). But, if nodes are allowed to continually update their composite values, then a quorum that appears at one moment could disappear in the next.

The balloting procedure used to solve that dilemma contains a large number of details, including both rules and heuristics, but at its core is surprisingly simple. First we will discuss the fundamental structure of balloting, then build up those details by progressively pointing out problems and their solutions.

Essentials of convergence

Intuitively, the first goal of a node that has obtained a composite value from nomination is to find a quorum that also has that composite value. Finding that quorum is called *preparing* a value. In other words, a composite value is prepared when it has passed through a round of federated voting and become confirmed. Any number of intermediate composites may come out of nomination before a quorum agrees on one, so to prevent nodes from getting stuck, we say a vote to prepare a composite value doesn't contradict any other vote to prepare a composite value. This means numerous composites can become confirmed prepared.

Once a composite is confirmed prepared, the node wants to *externalize* it, which means finalizing it for use as an output of the SCP slot. However, since there are many possible composite values, but only one may succeed in the end, we must *commit* one of the confirmed prepared values by performing federated voting on them. In fact, statements of the form 'commit the confirmed prepared composite value X' are voted for alongside statements 'prepare the composite value Y' (the same federated voting round). We define two critical rules.

1. Since only one composite can be finalized by the protocol, we define an ordering of composites. For now, imagine the ordering is based on an implementation-defined assessment of which composites are 'better' than others. For example, in MobileCoin composites could be ordered by transaction fee totals, ensuring new composite values are always ordered higher than old ones. This ordering scheme is a heuristic that encourages better composites to be finalized (in our naive pre-balloting model).

2. Statements to prepare a value X contradict statements to commit a value Y if $X > Y$.[23]

[23] The SCP whitepaper [115] says that preparing a value X 'aborts' all lower-ordered composites, however we believe the 'contradiction' terminology is a lot easier to understand, since it directly ties into our discussion of federated voting.

Importantly, we know from Section 10.2.2 that once a node has accepted a statement S, no other node in the network will accept a contradictory statement S'. In the context of balloting, once a node has accepted the statement 'commit Y', no other node will ever accept 'prepare X' if X > Y. Therefore Y is the highest value that can become prepared and is safe to externalize once it gets confirmed committed.[24]

It may seem like what we have described is adequate to guarantee convergence. However, the network can still get stuck if some nodes have voted to commit Y but other nodes voted to prepare X, leaving no full quorums able to use v-blocking sets to convince nodes on either side to change their minds. This can happen when some nodes see a full quorum accepting prepare(Y), so they confirm it and cast a vote for commit(Y), but part of that quorum voted for prepare(X) before they saw that prepare(Y) can be confirmed.

Resolving the stuck state

Even though the network might be stuck due to conflicts between prepare and commit votes, we do know that in this stuck state there is at least one value confirmed prepared, and hence the network has reached a point of convergence. Since it is plausible for a confirmed prepared value to be externalize-able, we can define two additional rules.

1. Instead of preparing and committing composite values directly, we define a *ballot* as a pair between a *ballot counter* and a composite value (called a ballot 'value' for short). New composite values obtained from nomination start at ballot counter 1 (for now). For example, a ballot value Y with counter 1 will be denoted Y:1.

   A ballot with a higher counter is always ordered higher than a ballot with a lower counter, and if two ballots' counters are the same, then ordering is defined by a comparison of their values. Moreover, two ballots with the same value never contradict each other, regardless of their counters.

   consensus/scp/src/core_types.rs *struct* `Ballot`

2. Whenever a node sees any ballot Y:n that is confirmed prepared, if that ballot is the *highest* confirmed prepared ballot, then the node will issue a vote to prepare Y:(n + 1).

These new rules lead to a number of observations.

1. Trivially, the two new rules do not violate the old rules. To exercise our understanding, consider these events that may occur.

[24] We can also point out that each node will only vote to commit one composite value at a time. To vote to commit a value, the node must confirm it is prepared, which entails accepting it is prepared. Therefore, if a node has voted to commit a value, then in order to accept it was prepared, the node must have dropped any lower-ordered votes to commit. Likewise, to vote to commit a higher value, the node must accept it is prepared, which requires dropping the current vote to commit.

   (a) If you have a vote for commit(Y:1), then you will also issue a vote for prepare(Y:2), because prepare(Y:1) must be confirmed.

   (b) If you confirmed prepare(Y:1), but have a vote for prepare(X:1) with X > Y, then you can't have voted commit(Y:1). Or perhaps you did vote commit(Y:1), but a v-blocking set accepting prepare(X:1) made you discard your vote commit(Y:1) and instead accept prepare(X:1) (which also qualifies as a vote for prepare(X:1)).

   (c) If you confirmed prepare(Y:1) and voted prepare(Y:2), then later confirmed prepare(X:1) and voted prepare(X:2), this implies prepare(Y:2) wasn't previously confirmed, because otherwise you would vote to commit(Y:2) which would contradict a vote for prepare(X:2) (setting aside the possibility of the commit vote being blocked by a prepare(N:n) with N:n > Y:2).

2. If commit(X:1) gets accepted by at least one node, but some nodes have voted to prepare(X:2), they won't get stuck because X:1 and X:2 don't contradict each other. If at a later point commit(X:2) gets accepted or even confirmed, it will not affect the value externalized (both ballots have value X).

3. Casting a vote to prepare(Y:2) for the highest confirmed prepared ballot Y:1, instead of all confirmed prepared ballots, is necessary in some cases because often nodes that confirmed prepare(Y:1) have also voted to commit(Y:1). Even if prepare(Z:1) was also confirmed, with Y > Z, since prepare(Z:2) contradicts commit(Y:1), those nodes can't vote for it. On the other hand, it might seem superfluous to add that sub-rule (that only the highest confirmed ballot should be bumped up with a new vote), since the contradiction rule will prevent votes to prepare(Z:2) anyway (for example).

   However, the 'issue new vote for highest confirmed prepared ballot' rule is more expansive because it also applies to nodes that have confirmed at least one ballot is prepared, but not voted to commit any ballots. This can happen if they voted to prepare a ballot ordered higher than any confirmed prepared ballots, preventing the nodes from voting to commit any of them.

   Suppose a node has voted to prepare(X:1), and confirmed both prepare(Y:1) and prepare(Z:1), with X > Y > Z. Now suppose (for the sake of argument) it votes for both prepare(Y:2) and prepare(Z:2). We know that prepare(Y:1) already was confirmed, so if enough time passes, all other nodes that confirmed it should also vote to prepare(Y:2), and then eventually confirm it (see an exception in the next observation). Since Y > Z, it is unlikely that a vote to commit(Z:n) will ever appear, since votes to prepare(Y:n) will in most cases be around to contradict it. Of course, that is only in most cases — if Z is lucky, it could win in the end. Therefore the part about 'highest confirmed prepared ballot' is a *heuristic* that simplifies the protocol.

4. If prepare(Y:1) was confirmed, then eventually all nodes in the network will also confirm it. Moreover, if enough time passes, then they will also vote to prepare(Y:2). However, that is only true if they do not instead decide to vote to prepare(X:2). Importantly, if any node

votes to prepare(X:2), it means prepare(X:1) has been confirmed. And, since X > Y, all nodes who confirmed both prepare(X:1) and prepare(Y:1) will vote for prepare(X:2) even if they previously voted to prepare(Y:2). In other words, since X:1 is superior to Y:1, X:2 is able to overwhelm Y:2 given enough time.

5. The previous observation gives us a hint about how these rules solve our previous problem of nodes getting stuck. Recall that nodes get stuck if they vote to prepare(X:1) before confirming prepare(Y:1), but other nodes confirmed prepare(Y:1) and voted to commit(Y:1), leaving them unable to vote for prepare(X:1), so prepare(X:1) is prevented from gaining enough votes to get accepted.

   With the new rules, eventually all nodes will vote to prepare(Y:2), thereby escaping the state of being stuck due to votes to prepare(X:1).

6. While increasing ballot counters can allow nodes to move past stuck states, it is still possible for ballot counters above 1 to get stuck in the same way. Suppose we have two ballots X:1 and Y:1 with X > Y. To get to the next ballot counter, imagine prepare(Y:1) is accepted by all nodes, but before any realize it is confirmed they also vote to prepare(X:1). First all the nodes realize prepare(Y:1) is confirmed and vote prepare(Y:2), and then a subset of nodes confirm prepare(X:1) and vote to prepare(X:2).

   Before the remaining nodes realize they can vote to prepare(X:2), they confirm prepare(Y:2) and vote to commit(Y:2). This is the same problem we had before, except now the counter is 2. To get past it, nodes have to vote to prepare(Y:3) after all of them confirm prepare(Y:2).

All those observations are great, and the reader may believe we have figured everything out by now. However, it is theoretically possible for the ballot counter to rise indefinitely with the stated rules, preventing the network from ever externalizing a ballot value.

Let us imagine two ballot values A and B, where A > B. These ballot values can 'ping-pong', neither able to overcome the other.

1. To start, suppose prepare(A:1) is voted for by a quorum, and then prepare(B:1) gets confirmed. The nodes that confirm prepare(B:1) will vote to prepare(B:2).

2. While prepare(B:2) is being voted for, prepare(A:1) gets confirmed. If any nodes had voted for commit(B:1), those votes will be discarded once v-blocking sets of nodes accepting prepare(A:1) cascade throughout the network.

3. Let there be a full quorum that votes for prepare(B:2) before realizing prepare(A:1) can be confirmed. None of those nodes may vote for commit(A:1) (although nodes outside that quorum may do so). On the other hand, they can still vote for prepare(A:2).

4. By the time prepare(B:2) gets confirmed, a full quorum has voted for prepare(A:2). As expected, prepare(B:3) gets voted for.

5. Now we see the pattern repeating. As prepare(B:3) gains votes, prepare(A:2) gets confirmed and prepare(A:3) votes appear.

6. While the pattern could go on forever, we will demonstrate how it can be broken. Let the votes for prepare(A:3) and prepare(B:3) accumulate. This time, votes for prepare(A:3) accumulate much faster than for prepare(B:3). Prepare(A:3) gets confirmed before prepare(B:3), and when the dust settles we see a full quorum voting for prepare(A:4), but no quorum for prepare(B:4) (perhaps some nodes voted prepare(B:4), but not enough for a quorum). This means prepare(A:4) is guaranteed to be confirmed, commit(A:4) will be accepted and confirmed, and A will be externalized.

It turns out the ping-pong problem can't be completely solved due to the asynchronous nature of SCP. Instead, we rely on heuristics to prevent it from impacting any real-world implementation of SCP.[25]

Synchronization heuristics

Fundamentally, ping-pong occurs when nodes issue votes for ballots with high counters even though the network is still working on lower-numbered ballots. The basic solution is for nodes to wait a while before issuing votes for ballots with higher counters. We define the following heuristics.

consensus/scp/src/slot.rs `maybe_set_ballot_timer()` `process_timeouts()`

1. If the highest ballot that a node has voted to prepare has counter N, then if the node sees a quorum whose highest ballots voted to prepare have counters $\geq$ N, the node will start a timer.

2. At the end of that timer, the node may issue a vote to prepare a ballot with counter N + 1, but not before (there is an exception, which we will point out later).

3. The timer duration rises as a function of N.

It is impossible for nodes to know if the network is stuck at any given ballot counter N, but also impossible to know if the network is ping-ponging. Relying on a steadily-increasing timer means if the network gets stuck, it will always 'try again' on a higher ballot counter, while if the network is ping-ponging, then it will approach a timer delay where ping-pong can be resolved automatically (since the appropriate delay is unknown a priori).[26] Moreover, always waiting for a full quorum before starting the timer helps keep nodes in line with each other, working on the same ballots at the same time.

[25] The reader may be disappointed to see we encountered an unsolvable problem after all the effort invested so far. However, it is a lot better to have a problem that can be addressed with heuristics than a problem with absolutely no solution like the network being at risk of getting stuck.

[26] If there are enough malicious nodes to block all quorums, those nodes could time their messages carefully to force the network into a perpetual state of ping-ponging. This is referred to as 'preemption' in the SCP paper [115]. Since a blocking set of malicious nodes has many ways to mess with a network using SCP (most of which we leave to the reader's imagination), we do not believe preemption represents an especially outstanding threat.

Miscellaneous optimizations

At this point, we have covered the conceptual foundation of SCP balloting. On top of the prior rules and heuristics, a litany of additional non-essential optimizations were defined in [115] and [24].

1. Once a node has confirmed a ballot is prepared (but not necessarily voted to commit it), it will no longer actively participate in nomination. In other words, the node will stop voting for, accepting, and confirming nomination statements, and won't update its nomination-derived composite value again.

   This is an optimization because it reduces the amount of composite values that can compete with each other during balloting. By itself, it is a trivially safe rule, because after a ballot is confirmed prepared, the network always has a path to externalizing something.

2. So far we have required that 'better' composite values be ordered higher than 'worse' composite values according to implementation-defined criteria. Perhaps it is more efficient not to have so strict a requirement for implementers of SCP. Frankly, it is not clear if the following rule is an optimization or just a complication.

   After the ballot timer expires, nodes *always* issue a vote to prepare a ballot with a higher counter. Nodes effectively store a 'global' ballot counter, which increments at the end of each timer period. The value for the new ballot gets selected according to the following priority list (highest priority first).

   consensus/scp/src/slot.rs `get_next_ballot_values()`

   (a) the highest confirmed prepared ballot's value
   (b) the current composite value from nomination
   (c) the highest accepted prepared ballot's value
   (d) the highest voted prepared ballot's value

   This 'optimization' complicates our conceptual model because instead of all ballots with counters > 1 representing ballot values confirmed prepared at the next lower counter, now the ballot stack can be polluted with apparently random ballots. It does, however, permit composite value ordering to be arbitrary, as ballots with new values obtained from nomination will always be ordered higher than old ballots (on a per-node basis).

3. Since in general the network moves to new ballot counters when lower counters are possibly stuck or not working, it is useful to define a catch-up mechanism for nodes that are at ballot counters lower than the network.

   If a node sees a v-blocking set with higher global counter values, then the node will update its local counter to the lowest value such that it is no longer blocked, and issue a new vote to prepare a ballot using that counter (with the value selected based on the prior optimization's priority list).

4. To reduce the size of messages sent between nodes, a declaration to vote for or accept the preparation of a ballot is also a declaration to do so for all ballots with the same value and lower counters.

5. If a node sees a ballot it can vote to commit, it may be able to vote to commit a range of ballots, since when a ballot is accepted prepared, all lower ballots with the same value are also accepted prepared. We will be discarding votes to prepare ballots in the next optimization, so there is a risk of casting a vote to commit (as part of a range of such votes) that is contradicted by a discarded vote to prepare a higher ballot with different value. As such, when issuing a new range of votes to commit a ballot, the bottom of that range must be $\geq$ the highest vote to prepare a ballot that has been cast up to that point in time (even if that ballot contains the same value as the vote-to-commit ballot).

   If votes were not discarded, then we could loosen that constraint by issuing all possible votes to commit that aren't contradicted by prior actual prepare-ballot votes or acceptances.

6. So far we have assumed nodes keep track of all ballots they have voted for, accepted, or confirmed (either to prepare or commit). We have also assumed each node has a complete picture of all that information for each other node (excluding votes to commit a ballot that are discarded due to v-blocking sets preparing a higher ballot with different value, or votes to prepare a ballot discarded due to v-blocking sets committing a higher ballot).

   Unfortunately, processing all that information over and over again entails a computational burden on nodes. Likewise, transmitting each node's full state (ballots voted for, accepted, and confirmed both to prepare and commit) to all other nodes each time it gets updated entails a high bandwidth cost that grows as the network gains nodes. It is beneficial, therefore, to minimize the amount of information necessary to execute SCP securely. However, discarding information runs the serious risk of introducing failure points if it allows the network to reach a state where nodes have forgotten too much and can't make progress.

   Surprisingly, it is only necessary for nodes to keep track of up to four pieces of information. Nodes store, and transmit to each other, the following.

   (a) **B**: The highest vote to prepare a ballot.

   (b) **P**: The highest ballot accepted prepared.

   (c) **PP**: The second-highest ballot accepted prepared (with a different value than **P**).

   (d) [**C**, **H**]: If the node has voted to commit a ballot, then **H** is the highest ballot voted to commit and **C** is the lowest ballot voted to commit (i.e. they represent a range of ballots with the same value).

   If the node does not have a vote to commit (either it has not yet voted to commit, or all prior commit votes were discarded), then **H** is the highest ballot confirmed prepared and **C** is a `null` ballot. Note that if **H** is not a vote to commit, then it is ignored by other nodes and only has significance to the local node.

   If the node has accepted a ballot committed, then [**C**, **H**] is the range of ballots accepted committed. Similarly, if the node has confirmed a ballot committed, then [**C**, **H**] is the range of ballots confirmed committed.

   There are a few preliminary observations we can make.

(a) When **C** is first set, it must be $\geq$ **B** according to the prior optimization. It is also easy to see that once **C** is set, it must be set to `null` before it can change to a different ballot (with different value or counter), because it only needs to change when contradicted by a higher ballot accepted prepared with different value.

(b) If **C** is set, meaning there is a vote to commit a ballot, then **P** will have the same value and **PP** will be less than **C**. This is because a vote to commit only appears when a ballot is confirmed prepared and there are no contradictory higher ballots that have been accepted prepared (or voted to prepare). **PP** exists so it is possible to know the lowest ballot counter **C** may have.

(c) There is an important detail to emphasize. After a node confirms a ballot is prepared and sets **H**, each time **B** is updated it will contain the same value as **H**. This aligns with the original rules we introduced for resolving stuck states.

We now briefly argue that the network cannot get stuck due to nodes forgetting too much information.

The network could plausibly get stuck if most of the network has stopped updating their nomination-phase composite value, but the available information contained in nodes' messages prevents progress. Nodes end nomination when they have confirmed a ballot is prepared. There are two cases to consider.

First, a node may confirm a ballot is prepared, but later on there are no longer any quorums that accept the ballot is prepared. Nodes drop their accepted prepared ballots when they accept a higher ballot prepared. Therefore if a confirmed prepared ballot is lost, there must be a higher ballot the network is working on.

Second, suppose all nodes in the network except one quorum have stopped making progress because they confirmed a ballot prepared that was dropped. Is it possible for this final quorum to reach a state where it can't make progress, dooming the entire network to stagnation?[27]

For this quorum to get stuck, part of it must end the nomination phase. Otherwise, the quorum could keep approaching convergence via nomination composites. Therefore we can assume some nodes will confirm a ballot prepared. There are two scenarios.

(a) The nodes could confirm a ballot prepared in conjunction with nodes outside the quorum under scrutiny. This would allow those nodes to get unstuck, unwinding the problem to a network state where more than one full quorum is not stuck.

(b) The nodes could confirm a ballot prepared within the quorum under scrutiny. For the quorum to get stuck, only some nodes can confirm the ballot prepared. Therefore the remaining nodes must drop their statements accepting the ballot prepared by accepting other higher ballots prepared.

[27] If the 'final quorum' has misbehaving nodes, then it is plausible for the network to get stuck. In that case, it is the responsibility of node operators to re-configure their quorum slices to remove bad nodes so the network can get unstuck.

i. If they accept higher ballots prepared based on the cooperation of outside nodes, this implies those outside nodes will get unstuck, allowing the network to progress.

ii. If they are able to accept higher ballots prepared based on votes cast by the quorum under scrutiny, then those votes *must also* be accompanied by statements accepting the ballot that was confirmed prepared by some nodes in the quorum. By the time the new higher ballot gets accepted prepared by a node, it will confirm the other ballot is prepared! Moreover, once a ballot is confirmed prepared, new votes to prepare a ballot always have the same value as the confirmed prepared ballot. Therefore a full quorum will issue votes to prepare the ballot confirmed prepared at a higher ballot counter, and the final quorum won't get stuck.

7. We saved the most dubious optimization for last. Given all the rules discussed so far, it is feasible for a node to accept a ballot is prepared that is higher than the ballot you wish to vote to prepare. There are two nuances here.

   (a) New votes to prepare a ballot **B** always have a higher counter than old votes to prepare a ballot **B**. This means if a ballot is accepted prepared based on a quorum of votes to prepare it, then any future votes to prepare a ballot will be higher than the ballot accepted prepared. Therefore a ballot accepted prepared will only be higher than **B** if it was accepted due to a v-blocking set.

   (b) Since ballots 'catch up' to other nodes whenever there is a v-blocking set at a higher ballot counter, any ballot accepted prepared will only be higher than **B** if its value is ordered higher than **B**'s value.

   Apparently it is considered more efficient for **B** to always be higher than any ballots accepted prepared, because SCP includes a 'clamp' on **P** and **PP**. When putting together an SCP message to send to other nodes, a node will reduce the counters of **P** and **PP** so that $\mathbf{B} \geq \mathbf{P} > \mathbf{PP}$. This effectively means the node will 'decline to notify' other nodes when it has accepted ballots prepared that are higher than **B**. Whether doing so is actually an efficiency gain or just meaningless is not at all clear.

   consensus/scp/src/slot.rs `out_msg()`

Balloting procedure outline

SCP contains a large number of rules, heuristics, and optimizations. Translating those into an implementation is not necessarily a trivial task. As such, the original paper [115] outlined an algorithm for balloting, which was elaborated on in an IETF draft [24] that discussed how it should be implemented. To close out this section, we will reproduce the algorithm outline, with minor changes introduced by the IETF draft. For more specific implementation details, the reader should consult the cited materials and MobileCoin's source code [71] (or the Stellar network's source code [158]).[28]

[28] The only large topic not covered in this chapter is quorum slice reconfiguration that occurs while a slot is being consensuated, since MobileCoin does not currently support it. Readers may consult [115] for a treatment of that

To organize the algorithm's content, four phases are defined: `NOMINATE`, `PREPARE`, `COMMIT`, and `EXTERNALIZE`. These phases are implemented as 'rounds', i.e. functions containing sequences of steps to take. `NOMINATE` and `PREPARE` may take place concurrently, although `NOMINATE` never lasts longer than `PREPARE` (it ends once a ballot has been confirmed prepared, as we have discussed).

consensus/scp/src/slot.rs `do_prepare_phase()` `process_timeouts()`

First is the `PREPARE` phase. The `NOMINATE` phase was discussed in Section 10.3.1. Note that the `PREPARE` and `COMMIT` phase rounds can be executed ad hoc (usually when receiving fresh messages from other nodes), and are not directly tied to ballot timers. Ballot timers only determine when the node should increment **B**'s counter.

1. Use the most recent messages from nodes in the network to see all the ballots that can be accepted prepared. If it is possible to increase **P**, do so. If **PP** can change, then change it (even if it decreases). If either **P** or **PP** contradict **C**, then set **C** to `null`.

2. Use network messages to see if any ballots can be confirmed prepared. If the highest such ballot is greater than **H**, then update **H**. The first time **H** is set, disable the `NOMINATE` phase.

3. If **C** is `null` and **H** is both higher than **B** and not contradicted by **P** or **PP**, then set **C** as low as possible. Doing so effectively means casting a vote to commit the range [**C**, **H**].

4. If the node can accept any ballot is committed, then set [**C**, **H**] to the lowest range of ballots that can be accepted committed. Set the value of **B** to match the ballot accepted committed. Also make sure **B**'s counter is $\geq$ **H**'s counter. If **B**'s value changed, then **B**'s counter should be higher than the prior **B**'s counter. Finally, set the phase to `COMMIT` and end the current `PREPARE` phase round.

5. Otherwise if **H** has been set (a ballot was at one point confirmed prepared), make sure **B**'s value matches **H**'s value, and **B**'s counter is $\geq$ **H**'s counter. If **B**'s value changes due to this step, then its new counter should be greater than its old counter.

   Note that in this and the previous steps, **B**'s counter can be incremented due to a change in value regardless of the ballot timer, a deviation from the heuristic we earlier defined. Furthermore, this step implements an important idea we discussed earlier, namely, voting to prepare the highest confirmed prepared ballot.

6. If there is at least one v-blocking set reporting **B**s with higher counters than the local node's **B**, then update **B**'s counter to the lowest number such that the node is no longer blocked. Either turn off the timer or reset it if appropriate (i.e. if there is a quorum at the new ballot counter), then redo the `PREPARE` phase round immediately to refresh the node's state.

subject. To summarize, active reconfiguration is only safe if quorum intersections are preserved in all permutations of quorum sets reported by nodes (assuming all befouled nodes are removed first). In other words, only if no combination of historical quorum sets from the network can be used to make quorums that don't intersect, is reconfiguration permissible. Lack of support for active reconfiguration may reduce nodes' ability to recover if they are befouled by unresponsive nodes or nodes sending illegal/dishonest messages. However, in practice it may be difficult to implement reconfiguration safely or efficiently due to the complexity of correctness. In MobileCoin, node operators must restart their nodes with a new quorum set to reconfigure.

Next is the `COMMIT` phase, which occurs after a ballot is accepted committed. In this phase, the values of **B**, **C**, and **H** do not change, and [**C**, **H**] represents the range of ballots accepted committed. **P** should be the highest ballot accepted prepared with the same value as **H**, and **PP** should be set to `null`.

consensus/scp/src/slot.rs `do_commit_phase()`

1. Using the latest messages from the network, update **P**'s counter if higher ballots with the same value are accepted prepared.
2. Use the latest messages from the network to update the counters in [**C**, **H**].
3. If a range of ballots with the same value as **H** can be confirmed committed, set [**C**, **H**] to that range of ballots, set the phase to `EXTERNALIZE`, and exit the current `COMMIT` phase round.
4. Otherwise update **B**'s counter so it is $\geq$ **H**'s counter.
5. If there is at least one v-blocking set reporting **B**s with higher counters than the local node's **B**, then update **B**'s counter to the lowest number such that the node is no longer blocked. Redo the `COMMIT` phase round immediately to refresh the node's state.

Last is the `EXTERNALIZE` phase. Once a ballot has been externalized, it can be used as the final output of the current slot. Note that ballot timers are not set during the `EXTERNALIZE` phase.

consensus/scp/src/slot.rs `do_externalize_phase()`

1. Using the most recent messages from the network, see what ballots can be confirmed committed. Increase **H**'s counter if possible. Doing so presumably keeps the node in alignment with the rest of the network if other nodes are still working on the slot.

10.4 Synchronizing nodes

So far we have imagined a network that is consensuating statements as if all nodes are more-or-less on the same page. Perhaps some nodes are still finalizing one slot when other nodes are working on the next one, but overall the nodes are collaborating actively. However, what if one node disconnects from the network for a period of time, and when it comes back, some nodes are several slots ahead of it? Or, what if a node is actively making slots, but discovers part of the network is externalizing contradictory slots (nodes in that part of the network have diverged from you)?

10.4.1 Catching up to the network

If a node is behind the network, naturally it will want to catch up. How can a node know when the network is actually ahead, as opposed to only some nodes claiming the network is ahead? As part of their normal operation, SCP nodes can use the following procedure to check if they are behind the network (on a lower slot index), and then catch up if necessary.

consensus/service/src/byzantine_ledger/worker.rs `tick()`

1. Node v checks if it is behind the network by identifying all other nodes working on a higher slot. If those nodes contain a v-blocking set, and (in combination with v) can form a full quorum, then the node probably needs to catch up.

   ledger/sync/src/network_state/scp_network_ledger/sync/state.rs `is_behind()`

2. Using those higher-slot nodes, find the highest slot agreed on by a v-blocking set (i.e. externalized by that set), and also by a full quorum. The v-blocking set convinces v to accept the slot, while the quorum convinces v to confirm it (all quorums seen by v include v).

   Finding the highest slot with a blocking set and quorum is an efficiency technique that assumes if multiple nodes externalize the same slot, all the lower slots they externalized must be the same. In other words, we assume there will also be a blocking set and quorum for all slots below the new highest slot. In MobileCoin, where slot contents (blocks) depend on previous slots (each block ID is a function of all earlier block IDs), this is an easy assumption to make.

   src/ledger_sync/ledger_sync_service.rs `attempt_ledger_sync()`

3. Request all slots up to the new highest slot from the nodes that are ahead. Externalize each slot if it appears valid. For MobileCoin, this means checking that each block's parent ID connects to the previous (all the way back to our local blockchain), that no key image duplicates appear, and that block root elements are as expected (reflecting the output set that existed after the previous block was published).[29,30]

   ledger/sync/src/ledger_sync/ledger_sync_service.rs `identify_safe_blocks()`

10.4.2 Healing divergences

Nodes only finalize statements in SCP when they are confident the network will also finalize those statements. This confidence is founded in three rules set by the federated Byzantine agreement model.

1. First, a node's basic goal is to reach agreement on statements with one of its quorum slices. As discussed early in this chapter, a quorum slice is a group of nodes in the network that the node trusts are well-behaved, and which the node would be satisfied to be in agreement with.

2. Second, all nodes seek agreement with their own quorum slices, so it is natural that agreement can only reliably appear when a full quorum of nodes reaches agreement.

3. Third, if nodes only finalized statements based on the first quorum of agreement they see, then the network would be unstable as nodes on the periphery of different quorums constantly get stuck. Instead, nodes are only willing to finalize statements when the rest of the network is also destined to finalize the same statements (via cascading v-blocking sets).

[29] In ledger synchronization, MobileCoin does not currently validate blocks' root elements.

[30] Recall that in MobileCoin, nodes always sign blocks they participate in validating (Section 9.5). Any block a node *doesn't* validate is not signed, which includes any blocks obtained when catching up to the network or resolving a fork.

However, those three rules leave no room for the question: what about network divergence that does occur? In other words, when a node externalizes some slots, they are externalized because the node is *confident* the network will also externalize them, and confident no nodes will externalize contradictory slots (e.g. fork a blockchain by adding different blocks at the same block height).

Nodes on opposite sides of an SCP network divergence have no automatic way to identify which side is 'canonical'. Each node is fully confident in the results it finalizes. The only way to resolve the problem is manual intervention by node operators.

Network divergence can only happen when non-intersecting quorums externalize different slots. This is the result of either quorum slice misconfiguration, or misbehaving nodes breaking nominal quorum intersections. Therefore node operators must decide if their quorum slices need to be improved. They must also decide if they are on the side of the network destined to survive the healing process. If not, they must discard unwanted slots and replace them with slots from the divergent network.[31]

In the case of MobileCoin, node operators can resolve unwanted forks by reconfiguring their quorum slices, rolling back their local copy of the blockchain to the block that appeared right before the fork (if their local copy needs to be discarded), and allowing the node software to resynchronize (Section 10.4.1) with the network using the reconfigured quorum slices.

[31] In Nakamoto-based blockchains, network forks are resolved automatically by each node selecting the chain fork with the highest cumulative difficulty (a variable related to the proof-of-work model).

Part II

Extensions

CHAPTER 11

Fog Service

As discussed in Section 4.3.1, for Bob to identify outputs he owns on the blockchain, he must perform (at least [168]) a Diffie-Hellman exchange on every txout public key that exists (compute $k_B^v * r_t K_t^{s,i}$). Moreover, this implies obtaining (e.g. downloading) a copy of every transaction output. In terms of bandwidth and computational effort, using this method to identify owned outputs is prohibitively expensive for weak devices with bandwidth constraints (e.g. mobile devices or hardware wallets) [79].

'Fog' is a service designed to address those problems by offloading the cost of scanning the blockchain to external servers, so users of the service can learn about their owned outputs efficiently. A naive approach would be for users to give copies of their private view keys to the service operator, however that would allow the operator to learn extensive information about the users' financial activities.

Instead, a Fog service is able to identify outputs owned by its users, and pass them along to those users, without the operator learning much more than the approximate number of outputs involved.[1] To accomplish that, Fog employs a combination of SGX enclaves and an ORAM (oblivious memory) data structure called an OMAP (oblivious map; see the implementation in [138]).[2]

[1] Fog's threat model can be found here: [70].

[2] The content in this chapter was sourced from [25], [129], and the MobileCoin team's implementation of Fog (actively in development and not released into open source at the time of writing this). Note that Fog could be abstracted away from MobileCoin and implemented in a generic way. We limit our exploration to the only existing implementation, which is MobileCoin-centric.

11.1 Fog-enabled addresses

At its core, the Fog system has two steps. First, a Fog provider identifies outputs owned by its users (Section 11.3). Second, its users obtain copies of those outputs (Section 11.4).

Fog providers identify outputs by looking at an *encrypted fog hint* stored in each output. A fog hint is simply the output recipient's public view key encrypted with a Fog 'ingress key' (Section 11.2). Ingress keys can change from time to time, so if Alice wants to send an output to Bob, she must contact his desired Fog provider to obtain an active/current key.

To that end, a standard address in MobileCoin includes identifying information about the address-holder's desired Fog provider. Such an address is formatted like this:

account-keys/src/account_keys.rs *struct* PublicAddress

[public view key][public spend key][fog url][fog report id][fog signature]

The Fog-related fields are as follows.

- **fog url**: An identifier for locating and contacting the Fog service, along with the method for locating it (e.g. the http/https protocols). It has the format 'method://identifier'.[3]
- **fog report id**: If an address-holder's Fog provider is operating multiple distinct `fog-ingest` enclaves, this identifier indicates which enclave's ingest key to use when constructing an output for the address-holder.[4]
- **fog signature**: A 'root' key belonging to the Fog provider is signed with the address-holder's public view key. As we will see, this helps ensure encrypted fog hints are only readable by Bob's chosen provider even in the case of a secure enclave breach.

How Bob chooses his Fog provider and obtains the Fog components of his address are implementation details not pertinent to this discussion.

11.2 Sending outputs from Alice to Bob

For Alice to send an output to Bob via Fog, she must add an encrypted fog hint to the output. That hint is Bob's public view key encrypted with a Fog ingress key. Clearly, if his Fog provider knows the private ingress key, they can trivially learn which outputs are owned by Bob.

Rather, the private ingress key lives in a `fog-ingest` SGX secure enclave. To give transaction authors confidence they are only creating encrypted fog hints readable by a secure enclave, fog ingress keys sent to authors must have corresponding `Attestation Verification Reports` (`AVR`s, recall Section 8.2.6).[5]

[3] A URL (Uniform Resource Locator) is a *locator* that tells you how to locate a resource. URLs are a subset of so-called URIs (Uniform Resource Identifiers). [121]

[4] As we will see, there isn't much point for a Fog provider to have multiple `fog-ingest` enclaves, so in practice most Fog providers are likely to only use one 'fog report id'. However, the id may allow some providers to offer 'early access' to Fog service upgrades. Early access users could test new Fog features or infrastructure before they are rolled out to the main user-base.

[5] Alice could conspire with Bob's Fog provider to use a non-enclave key for the encrypted fog hint. However,

11.2.1 Ingress keys

*[MC-fog] fog/ingest/enclave/impl/src/lib.rs `enclave_init()` sgx/report-cache/untrusted/src/lib.rs `start_report_cache()`

Ingress keys, denoted $K_{ingress} = k_{ingress}G$, are Ristretto points generated when a `fog-ingest` enclave is initialized. The private ingress keys are encrypted and stored locally for re-use when an enclave is restarted.[6] They can also be transmitted to 'back-up' enclaves, which are 'peer' enclaves with the same `MRENCLAVE` value that can be activated in case the primary enclave has a problem.

The `fog-ingest` sub-service (i.e. the host process that owns the enclave) cooperates with its enclave to produce an `AVR` using $K_{ingress}$ as the report content. It is realistic to assume SGX secure enclaves can and will be breached (even if infrequently, with problems repaired quickly), so ingress `AVR`s are signed with a key belonging to the Fog service operator. This Fog key is in turn signed with another Fog key, and so on in a 'certificate chain' up to a so-called 'certificate root key'. The 'fog signature' in Bob's address is a signature on that root key.[7]

Since enclaves are periodically updated, ingress keys and their `AVR`s become invalid as time passes. As such, '`pubkey_expiry`' values are attached to ingress `AVR`s (but not part of the report content).[8] A `pubkey_expiry` is the anticipated block height where the ingress key will expire. Practically speaking, this means each transaction's tombstone block should be less than the lowest `pubkey_expiry` of its Fog-using recipients' ingress keys.

transaction/std/src/transaction_builder.rs `impose_tombstone_block_limit()`

The ingress key `AVR`, `AVR` Fog signature, certificate chain, and `pubkey_expiry` are stored in anticipation of requests from transaction authors.

11.2.2 Acquiring an ingress key

Alice can use the 'fog url' in Bob's address to contact his Fog provider.[9] She receives a set of ingress key `AVR`s (with associated material) for all the `fog-ingest` enclaves being run by the provider.[10] It turns out those `AVR`s also have a 'fog report id' attached (matching of `AVR` to 'fog report id' is up to the Fog provider's discretion). Alice chooses the `AVR` corresponding to the 'fog report id' in

fog/report/validation/src/lib.rs `get_fog_pubkey()`

*sgx/service/src/lib.rs `seal_data()`

mobile-coind/src/payments.rs `build_tx_proposal()`

Alice has plenty of other ways to reveal Bob as a recipient of her transaction, so conspiring in that way is not a specific threat Bob should worry about. The bigger danger is Alice's wallet software leaking information to the Fog provider or another party, unbeknownst to Alice or Bob.

[6] Specifically, private ingress keys are encrypted with an SGX Seal key using `KEYPOLICY.MRENCLAVE` set to `true` (recall Table 8-3 footnote 22), so only enclaves on the same machine with the same measurement can decrypt it.

[7] A certificate chain is used instead of a single Fog service key for easier key management. The root key can be stored in a very secure location, while intermediate keys and final signing keys can be managed in more convenient ways, and discarded easily without invalidating users' fog signatures (which should have an indefinite shelf-life).

[8] Ingress key `pubkey_expiry`s are defined by the Fog operator and can be altered at any time, although they should only ever be *increased*, as decreasing them may cause tombstone blocks set before the decrease to be so high that a transaction is added to the blockchain after an ingress key has expired and been discarded.

[9] 'Self-spends', or outputs that are addressed to oneself, are treated, in terms of Fog, the same as outputs addressed to others. In other words, if Alice sends an output to herself (e.g. a change output), she will contact her own Fog provider with the same procedure used to contact Bob's provider, and encrypted fog hint construction proceeds in the same way as well.

[10] Within the Fog service is a so-called `fog-report` sub-service which keeps track of `AVR`s and passes them along to requesters.

Bob's address.[11,12]

By verifying the `AVR`, the `AVR` Fog signature, the certificate chain, and Bob's address's fog signature, Alice can be sure the `AVR`'s ingress key was either produced by a secure enclave or at the very least belongs to Bob's chosen Fog provider (who he presumably trusts to some extent).

11.2.3 Encrypted fog hints

Once Alice has a Fog ingress key $K_{ingress}$, creating the encrypted fog hint is straightforward.

[MC-tx] src/fog_hint.rs `encrypt()`

1. Generate a random scalar $r \in_R \mathbb{Z}_l$.
2. Compute
$$r * G$$
$$r * K_{ingress}$$
3. Encrypt Bob's view key K_B^v with $rK_{ingress}$[13]
$$\texttt{enc}[rK_{ingress}](K_B^v)$$
4. Assemble the full encrypted fog hint[14]
$$\{rG, \texttt{enc}[rK_{ingress}](K_B^v)\}$$

If Alice wants to send an output to Carol, but Carol doesn't have a Fog-enabled address, then Alice can make a fake encrypted hint so Carol's output appears the same as all other outputs. One approach is to generate a random fake ingress key and encrypt an empty byte-field.[15,16]

[MC-tx] src/encrypted_fog_hint.rs `fake_one-time_hint()` crypto/box/

[11] It would seem like the 'fog url' could be used to communicate which `AVR` the Fog service should send Alice. However this would leak to the Fog provider some information about Alice's intended recipient.

[12] The Fog provider may be able to maliciously provide Bob with a unique fog report id and somehow leverage it to learn information about the outputs he owns. One way to detect this would be creating a bunch of new Fog addresses and seeing if they are all assigned the same (small) set of fog report ids.

[13] Encrypted fog hints rely in practice on a custom encryption scheme, described in [127], that is similar to ECIES [131] but implemented for the Ristretto abstraction. Basically the scheme takes a message $\mathtt{m}$ and public key K as inputs, generates a temporary key r, produces a shared secret rK, uses the shared secret to encrypt the message, creates a hash (known as a 'MAC') of the message and shared secret (which can be used to check if message decryption succeeded), and outputs rG, the MAC, and the encrypted message.

[14] An encrypted fog hint is 84 bytes, which includes the 32 byte key rG, and 52 bytes for the encrypted view key (16 bytes for the MAC, 34 bytes for the encrypted view key plus 2 bytes of padding, and 2 versioning bytes for the encryption scheme).

[15] As noted in footnote 6 of Chapter 4, there is no absolute requirement for wallet implementations to conform with transaction-construction standards like one-time addresses or encrypted fog hints. In the case of one-time addresses, non-conforming wallets are likely to create outputs completely unusable by other wallets. However, it may be possible to design wallets that eschew support for Fog, since the blockchain can always be scanned in the normal fashion. Such wallets should reject attempts to send funds to Fog-enabled addresses, to ensure they only create outputs for recipients that don't use/rely on Fog.

[16] The encrypted fog hint provides a convenient way to mitigate the Janus attack (recall Chapter 4 footnote 13). If instead of randomly generating the txout private key r_t it was set equal to the encrypted hint's r value, then MobileCoin users could compute $k^v(k^s + \mathcal{H}_n(k^v, i)) * rG \stackrel{?}{=} r_t K_t^{s,i}$ to check if the txout public key provided in a given output they receive was produced by the same subaddress spend key that was used to create the output's one-time address K_t^o. This approach has not been implemented in any form at the time of writing this.

11.3 Processing the blockchain

A Fog provider identifies its users' outputs by passing all transaction outputs from the blockchain into a `fog-ingest` enclave, which attempts to decrypt the fog hints. That enclave produces records of each output that can be easily recovered by their owners.

11.3.1 Decrypting fog hints

On the surface, identifying Fog users' outputs seems fairly trivial. Transaction outputs are passed into a `fog-ingest` enclave, which uses its private ingress key $k_{ingress}$ to decrypt the outputs' encrypted fog hints. If decryption succeeds (i.e. the hint's MAC matches a value computed from the decrypted message) and the decrypted value is a valid Ristretto point, then that point must be the output recipient's public view key.[17]

However, decrypting hints takes place in a secure enclave owned by a perhaps-not-trustworthy Fog operator. That operator has full visibility on the material passed in and out of an enclave, and the timing of those events. If there is an observable difference in how outputs are treated (i.e. between outputs that are and aren't owned by Fog users), then the Fog operator can use that difference as the basis for analyzing his users' behavior [25].[18]

To prevent the decryption of fog hints from leaking any timing information, they are decrypted in 'constant time'. This means the time it takes the decryption algorithm to execute is not a function of the algorithm's inputs (i.e. it's a 'data oblivious' algorithm [126]).

(E) [MC-tx] src/fog_hint.rs `ct_decrypt()`

11.3.2 Making outputs discoverable

It is not realistic to maintain ingress keys indefinitely. In the first place, as noted in Chapter 8 footnote 28, enclave secrets can't be safely migrated between enclave versions, while the SGX enclave technology receives periodic updates. This means once the owner of an output has been identified, the output needs to be stored in a permanent form so it can be recovered at any time (or even multiple times if necessary).

The basic solution adopted is for each output to be encrypted using the decrypted fog hint key (usually the recipient's public view key by convention), and emitted from the `fog-ingest` enclave tagged with a 'fog search key' (based on an 'egress key') that the recipient can recompute privately later on (Section 11.4).

[17] Technically any public key whose private key is known by the recipient can be used in fog hints, however using the public view key is a convention set by the initial implementation of Fog.

[18] Performing analyses based on evidence gained from indirect observation of a computer process is known as a 'side channel' attack [133]. If outputs can be categorized by Fog, then a Fog operator who learns the identity of a transaction's author (e.g. by tracing the IP address used to submit a transaction to the network) may gain insight into that author's recipients. Other insights may be possible as well, but by eliminating the side channel itself, their pursuit is rendered futile.

Egress keys

*[MC-fog] fog/ingest/enclave/impl/src/lib.rs `new()`

Each time a `fog-ingest` enclave is started up, it generates a unique 'egress key' k_{egress},[19] which is just an Ed25519 private key, and sends out a so-called `RngRecord` to its host process for permanent storage. The record contains $k_{egress} * G$, the first block processed by the enclave after that egress key was created, and the current 'rng algorithm' used to create fog search keys.

[MC-fog] fog/sql_recovery_db/src/lib.rs `new_ingest_invocation()`

Egress keys live shorter lives than ingress keys. In particular, this is because they must be recreated each time an enclave is started up (that can't be saved or sent to peer enclaves).[20] They are also occasionally recreated within a given enclave instance.

Fog search keys

*[MC-fog] fog/ingest/enclave/impl/src/lib.rs `attempt_ingest_txs()`

After a fog hint has been decrypted, the `fog-ingest` enclave creates a shared secret between the decrypted key and the egress key. Since this must be constant time, if decryption fails, the enclave will make a shared secret with a randomly generated key instead (it is generated before decrypting the fog hint so its creation can't be used for timing analysis). The final shared secret we denote $k_{egress} * K_{hint}$, where K_{hint} could be a real or fake fog hint key.

To accommodate users who may receive an arbitrary number of outputs to the same hint key, fog search keys depend on a counter that increments each time the fog-ingest enclave encounters a given successfully-decrypted hint key. These counters are maintained in an ORAM hash map (similar to a table) called an OMAP, which is owned by the `fog-ingest` enclave.

*[MC-fog] fog/ingest/enclave/impl/src/rng_store.rs `new()`

OMAPs are designed so accessing and modifying entries is constant time.[21] `Fog-ingest` enclaves construct a new OMAP for counters each time a new egress key is created. The shared secret $k_{egress} * K_{hint}$ is used to locate counters in the map, so even if a `fog-ingest` enclave operator manages to read the contents, he won't necessarily learn much about output recipients.

[MC-fog] fog/kex_rng/src/versioned/kexrng-20201124.rs `prf()`

Using the counter value obtained from the map (it is a fake/arbitrary value if K_{hint} is fake), an output's fog search key FSK is simply:[22,23,24]

[19] 'Ingress keys' process Fog inputs (outputs from the chain), while 'egress keys' help create the `ETxOutRecords` emitted from Fog.

[20] If an egress key could be reused by multiple enclave instances, then it could be used as part of a replay attack by the enclave operator. The operator could feed fake outputs addressed to Bob to a hidden enclave in order to see what fog search keys are produced, then watch the fog search keys emitted by a real `fog-ingest` enclave (using the same egress key) to figure out which real on-chain outputs are owned by Bob (the operator can correlate outputs sent into a `fog-ingest` enclave with `ETxOutRecord`s sent out from the enclave).

[21] We will not explore OMAPs in any depth. Suffice it to say for our purposes that an OMAP is a hash map that can be accessed and modified in constant time. Moreover, the structure that MobileCoin uses employs an encryption scheme so that OMAPs created by an enclave can use memory outside the `ELRANGE`, enabling arbitrarily large structures to be set up and used without leaking any information to the enclave owner. The implementation is open sourced at [138].

[22] The fog search key hash includes the 'fog search algorithm' version. This may not be strictly necessary, however it forces wallet developers to keep track of changes to the Fog search key algorithm (which may in reality never change).

[23] The hash algorithm used outputs 64 bytes, but only 16 bytes are considered necessary for fog search keys to be secure.

[24] Since fog search keys are based directly off the shared secret between an egress key and view key by convention,

$$FSK = \mathcal{H}(\text{algo version}, k_{egress} * K_{hint}, \text{counter}).\text{first\_16\_bytes}$$

The counter is incremented and put back in the table (only if K_{hint} is real), or a new OMAP entry is added if K_{hint} hasn't been encountered before (again, only if K_{hint} is real).[25,26] In this way, all of a user's outputs' fog search keys created while an egress key is active can be recomputed from the user's private view key, the public egress key, and incrementing a counter from 0.

Encrypted output records

To prepare an output for easy use by its owner, a `TxOutRecord` structure is created. That structure contains the output (sans encrypted fog hint), its on-chain index (i.e. its index in all the `TxOut`s ever created), its block's index, and that block's timestamp (to indicate when the output was created). The `TxOutRecord` is encrypted with the same scheme used to encrypt fog hint keys (recall footnote 13). This time the encryption key is K_{hint}, so the recipient can decrypt it later (Section 11.4).

*[MC-fog] fog/ingest/enclave/impl/src/lib.rs `attempt_ingest_txs()`

The fog search key FSK is placed alongside that encrypted structure in an `ETxOutRecord`, which is emitted from the `fog-ingest` enclave for permanent storage. One `ETxOutRecord` is created for every `TxOut` sent into the enclave, so the enclave operator cannot discern which ones belong to individuals using his service, and which ones do not.

[MC-fog] fog/sql_recovery_db/src/lib.rs `add_block_data()`

11.3.3 Missed blocks

Astute readers may have noticed a critical hole in the Fog system as described. An output can only be discoverable if (A) Alice submits it to the network with a fog hint encrypted by an active ingress key and (B) that fog hint is decrypted by the `fog-ingest` enclave using that ingress key. If requirement (A) or (B) isn't met, the `fog-ingest` enclave will output a junk `ETxOutRecord`.

It is ultimately Alice's responsibility to make sure the outputs she creates can be found by her intended recipients, so requirement (A) is out of scope for the design of Fog (aside from facilitating ingress keys with reasonable `pubkey_expiry`s; also, see footnote 27).

Normally, `fog-ingest` enclaves actively scan new outputs as they are added to the blockchain, so if Alice submits a transaction soon after acquiring an ingress key, the `fog-ingest` enclave that processes it should use the appropriate private ingress key. However, this is only in the ideal scenario. If the enclave crashes, its machine dies, or there is a Fog service-wide outage,

[MC-fog] fog/ingest/server/src/controller.rs `process_next_block()`

it is not possible to create a FSK chain that includes outputs from multiple subaddresses (note that a non-standard implementation could use the same fog hint key for multiple subaddresses). In other words, Fog is a service that only collects outputs discoverable by individual addresses and subaddresses by default.

[25] Note that this means the counter OMAP can only contain entries corresponding to unique keys found while a given egress key is active.

[26] If the counter OMAP gets full, then no more unique entries can be added. In that scenario, the existing OMAP is deleted, a new one is set up, and the `fog-ingest` enclave creates a new egress key (and emits it in an `RngRecord`). This strategy is functionally equivalent to restarting the enclave.

then Alice's transaction may not be properly examined. These potentialities are handled with a defense-in-depth approach.[27]

1. Basic lifetime: Ingress keys are generated when `fog-ingest` enclaves start up and are stored in temporary memory (the `ELRANGE`) until deprecated or the enclave is shut down.

2. Local storage: `Fog-ingest` enclaves encrypt ingress keys with an SGX Seal key when they are created, and save them on the local machine so they can be recovered if the enclave is restarted (only an enclave with the same `MRENCLAVE` value on the same SGX-enabled machine can recover it).

3. Peer enclaves: Ingress keys can be sent to 'peer' enclaves (enclaves with the same `MRENCLAVE` value), which act as back-ups in case the main enclave's machine has a critical failure.

4. Last resort: In the worst case scenario where an ingress key is completely lost/unrecoverable, yet there exist blocks within that key's expected lifetime (i.e. younger than its creation and older than its expiration) whose outputs haven't been processed into `ETxOutRecords` by that key, those 'missed blocks' must be manually scanned by users for them to identify all owned outputs therein. The Fog system is designed so its operators can identify missed blocks and send the outputs they contain to users.

*[MC-fog] fog/ingest/enclave/impl/src/lib.rs `enclave_init()`

[MC-fog] fog/ingest/server/src/controller.rs `sync_keys_from_remote()`

[MC-fog] fog/ingest/client/src/lib.rs `report_missed_block_range()`

11.4 Obtaining owned outputs from Fog

Obtaining owned outputs from Fog is fairly straightforward.

1. Contact the `fog-view` sub-service (it's not an enclave, just a normal server) and receive copies of all `RngRecord`s containing public egress keys that were active over the range of blocks you wish to scan, along with any missed blocks in that range.[28] The `fog-view` sub-service will also provide an `AVR` for communicating with a `fog-view` enclave that it owns.

2. Using the public egress keys in those `RngRecord`s, compute fog search keys with counters starting at 0. It is important to start by computing at least two fog search keys from each egress key (counter values '0' and '1').[29]

[MC-fog] fog/view/protocol/src/polling.rs `poll()`

[MC-fog] fog/view/protocol/src/user_rng_set.rs `ingest_tx_out_search_results()`

[27] To transition between enclave versions, the following procedure is feasible: a) start a new-version `fog-ingest` enclave (it generates a new $K_{ingress}$), b) stop making the old ingress key available to transaction authors (i.e. its `AVR`) and start making the new one available, c) once the old ingress key's highest `pubkey_expiry` is reached (the Fog service provider might choose to periodically raise the expiry block of active ingress keys, so only the highest expiry should be used) shut down the old-version `fog-ingest` enclave. For wallets to properly verify new `AVR`s, they likely have to be updated with each update to Fog enclaves. A similar procedure can be used to rotate ingress keys (i.e. deprecate an old key and start using a new one). Rotation can be useful for reducing the cost of a given ingress key being compromised (i.e. reduce the number of outputs' fog hints a single ingress key can decrypt).

[28] If your Fog service provider uses multiple 'fog report ids', then you may receive a set of `RngRecord`s corresponding to the report id in your public address (recall Section 11.1). In other words, a different set than might be received by a user with a different report id.

[29] We must use the 'full' private view key $k^{v,i}$ instead of base key k^v, because `RngRecords` use a shared secret between $k_{ingress}G$ and $K^{v,i}$.

$$FSK = \mathcal{H}(\text{algo version}, k^{v,i} * k_{egress}G, \text{counter}).\text{first\_16\_bytes}$$

3. Send the fog search keys to the `fog-view` enclave. This enclave has an OMAP loaded with `ETxOutRecord`s passed to it by the Fog operator, where fog search keys are used as the hash map lookup keys (hence the name 'fog search keys'). The enclave will look up any records corresponding to fog search keys sent by the user, and send them back.

   To prevent the Fog operator from inferring exactly how many outputs the user actually owns, the `fog-view` enclave is designed to be constant-time and constant-bandwidth. In part, this means it will send back fake `ETxOutRecord`s for any fog search keys that don't have entries in the OMAP. Note that the operator can still estimate the maximum number of outputs the user owns based on the overall volume of data transmitted between their `fog-view` enclave and the user.

   *[MC-fog] fog/view/enclave/impl/src/lib.rs `query()`

4. Alongside `ETxOutRecord`s returned to the user are 'result codes' indicating if the records correspond with OMAP lookup matches or misses. If all returned records from a given fog-search-key-chain are matches (regardless of if they can be decrypted or used in any way), then it's possible the OMAP contains matches for higher counter values (which may be usable). The user must continue making requests until the enclave returns a 'miss'.[30,31]

   A user only stops making requests for a chain when not all records returned are matches. This is why the minimum chain-request size should be two. If the size was one, it would be clear when a user doesn't own any outputs processed by a given egress key. By starting with two search keys, failure to make a second request means the user owns either one or zero outputs processed by the relevant egress key, which is ambiguous to the Fog operator.

   *[MC-fog] fog/view/enclave/impl/src/e_tx_out_store.rs `find_record()`

   [MC-fog] fog/view/protocol/src/polling.rs `poll()`

5. The user can decrypt the `ETxOutRecord`s he receives to reveal the outputs that he owns. Of course, he should also perform the usual steps for examining outputs to make sure he really owns them (computing the sender-receiver shared secrets, checking the one-time addresses [Section 4.3.1], and decrypting the amounts [Section 5.3]).

   [MC-fog] fog/view/protocol/src/user_rng_set.rs `ingest_tx_out_search_results()`

In the end, users obtain their owned outputs and Fog operators know few if any of the details (assuming either SGX secure enclaves are not breached, or Fog operators do not exploit any vulnerabilities that may be discovered).[32]

[30] If the `fog-view` enclave's `ETxOutRecord` OMAP is somehow corrupted, then it is conceivable for a 'match' to exist for a higher counter value than a 'miss'. However, there is no easy way for the user to detect this, so it is simplest to assume that a 'miss' ends the chain.

[31] Since the user may not know how many `ETxOutRecords` they will ultimately receive, it may be useful to create progressively larger quantities of fog search keys with each request, to reduce the total number of requests made. In the MobileCoin implementation, the request size is doubled with each iteration (starting at 2).

[32] If an SGX enclave breach does occur, then the Fog operator may be able to learn the private ingress key created by any `fog-ingest` enclave with an exploitable SGX enclave version (ingress keys can be saved in anticipation of future breaches, and old enclave software (re)started at any time). A leaked private ingress key can decrypt all fog hints encrypted with that key, effectively revealing the true recipients of those hints' outputs. This risk can be avoided by manually scanning all on-chain outputs instead of using Fog, at the cost of time and downloading all those outputs.

[MC-fog] fog/view/protocol/src/polling.rs `poll()`

11.5 Accessing blockchain information

In other RingCT-based cryptocurrencies such as Monero, users are only able to access blockchain data by running their own node (which provides them with a full copy of the blockchain), or making requests to node operators. Running a node is not a requirement all users are willing to accept, downloading large subsets of blockchain data is expensive and tedious, and requesting specific information from node operators is a privacy risk.

The Fog sub-service known as `fog-ledger` is designed to alleviate those problems. In short, a `fog-ledger` enclave can store OMAPs containing on-chain key images, a full and continuously-updated output Merkle tree for assembling membership proofs, and all on-chain outputs. Users can make direct requests to this enclave to privately learn information about the blockchain.[33]

11.5.1 Checking for spent outputs

The main Fog system is designed to connect users with their owned outputs, but has no way of knowing when those outputs are spent. Instead, output recipients must compute their outputs' key images, then check if those key images have appeared in the blockchain.

This is made simple by the `fog-ledger` sub-service. Bob can send any key image to the `fog-ledger` enclave, which will check if the key exists in its key image OMAP and inform Bob of the result.

11.5.2 Acquiring membership proofs

To construct a new transaction, the transaction author must include membership proofs that show outputs spent by the transaction appear in the blockchain (recall Chapter 6). These proofs can only be created from a full view of the blockchain.

MobileCoin has the ambitious design goal of making the on-chain transaction graph completely unknown to all observers. In keeping with that, obtaining membership proofs should not leak to Fog operators which ring members a user wishes to include in their transactions. To that end, the `fog-ledger` enclave can maintain an OMAP of the continually-updating output Merkle tree. Requests for membership proofs at specific on-chain output indices can be resolved obliviously using that OMAP.

11.5.3 Getting copies of outputs

How much needs to be said? The `fog-ledger` enclave can store `TxOuts` in an OMAP and deliver them to users on request (i.e. given on-chain output indices).

[33] The `fog-ledger` service is still in development at the time of writing this, so exact implementation details may differ from what is described to some extent. Readers should note that the `fog-ledger` service prototype does not use OMAPs, so all queries are visible to the Fog operator.

Bibliography

[1] Abelian Group. Brilliant.org. https://brilliant.org/wiki/abelian-group/ [Online; accessed 10/03/2020].

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

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

[4] Extended Euclidean Algorithm. Brilliant.org. https://brilliant.org/wiki/extended-euclidean-algorithm/ [Online; accessed 12/08/2020].

[5] Fermat's Little Theorem. Brilliant.org. https://brilliant.org/wiki/fermats-little-theorem/ [Online; accessed 12/02/2020].

[6] Firmware. https://simple.wikipedia.org/wiki/Firmware [Online; accessed 11/10/2020].

[7] Intel(R) SGX Reference Launch Enclave. https://github.com/intel/linux-sgx/blob/master/psw/ae/ref_le/ref_le.md [Online; accessed 12/01/2020].

[8] Software Guard Extensions: Attacks. https://en.wikipedia.org/wiki/Software_Guard_Extensions#Attacks [Online; accessed 11/09/2020].

[9] Transport Layer Security (TLS). IEEE. https://site.ieee.org/ocs-cssig/?page_id=658 [Online; accessed 12/01/2020].

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

[11] What is a premine? https://www.cryptocompare.com/coins/guides/what-is-a-premine/ [Online; accessed 06/11/2018].

[12] XOR – from Wolfram Mathworld. http://mathworld.wolfram.com/XOR.html [Online; accessed 04/21/2018].

[13] 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].

[14] Fungible, July 2014. https://wiki.mises.org/wiki/Fungible [Online; accessed 03/31/2020].

[15] 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].

[16] Base58Check encoding, November 2017. https://en.bitcoin.it/wiki/Base58Check_encoding [Online; accessed 02/20/2020].

[17] Alexandre Adamski. Overview of Intel SGX - Part 2, SGX Externals, August 2018. https://blog.quarkslab.com/overview-of-intel-sgx-part-2-sgx-externals.html [Online; accessed 12/01/2020].

[18] 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.

[19] 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].

[20] Jean-Philippe Aumasson, Willi Meier, Raphael Phan, and Luca Henzen. *The Hash Function BLAKE*. Springer Publishing Company, Incorporated, 2014.

[21] Jean-Philippe Aumasson and Luis Merino. SGX Secure Enclaves in Practice: Security and Crypto Review, July 2016. https://www.blackhat.com/docs/us-16/materials/us-16-Aumasson-SGX-Secure-Enclaves-In-Practice-Security-And-Crypto-Review-wp.pdf [Online; accessed 10/27/2020].

[22] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2: simpler, smaller, fast as MD5, January 2013. https://blake2.net/blake2.pdf [Online; accessed 10/06/2020].

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

[24] N. Barry, G. Losa, D. Mazières, J. McCaleb, and S. Polu. The Stellar Consensus Protocol (SCP) [IETF DRAFT], June 2018. https://tools.ietf.org/pdf/draft-mazieres-dinrg-scp-04.pdf [Online; accessed 03/04/2021].

[25] Chris Beck and koe. Privacy Properties of MobileCoin Fog, 2020. Unpublished manuscript draft.

[26] 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].

[27] 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].

[28] 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].

[29] bitcoin repository. Hierarchical Deterministic Wallets [BIP: 32]. https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki [Online; accessed 04/05/2021].

[30] bitcoin repository. Mnemonic code for generating deterministic keys [BIP: 39]. https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki [Online; accessed 04/05/2021].

[31] bitcoin repository. Multi-Account Hierarchy for Deterministic Wallets [BIP: 44]. https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki [Online; accessed 04/05/2021].

[32] 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].

[33] Dan Boneh and Victor Shoup. A Graduate Course in Applied Cryptography. https://crypto.stanford.edu/~dabo/cryptobook/ [Online; accessed 12/30/2019].

[34] Ernie Brickell and Jiangtao Li. Enhanced Privacy ID from Bilinear Pairing. Cryptology ePrint Archive, Report 2009/095, 2009. https://eprint.iacr.org/2009/095 [Online; accessed 12/01/2020].

[35] 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].

[36] 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].

[37] 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].

[38] Grin community. Introduction to Mimblewimble and Grin, September 2020. https://github.com/mimblewimble/grin/blob/master/doc/intro.md [Online; accessed 01/06/2021].

[39] Victor Costan and Srinivas Devadas. Intel SGX Explained. Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint.iacr.org/2016/086 [Online; accessed 10/27/2020].

[40] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Secure Processors Part II: Intel SGX Security Analysis and MIT Sanctum Architecture. Foundations and Trends in Electronic Design Automation, 2017. https://people.csail.mit.edu/devadas/pubs/part_2.pdf [Online; accessed 12/01/2020].

[41] dalek cryptography. Bulletproofs. https://doc-internal.dalek.rs/bulletproofs/index.html [Online; accessed 03/02/2020].

[42] dalek cryptography. bulletproofs [library]. https://github.com/dalek-cryptography/bulletproofs [Online; accessed 10/06/2020].

[43] dalek cryptography. curve25519-dalek [library]. https://github.com/dalek-cryptography/curve25519-dalek [Online; accessed 10/06/2020].

[44] dalek cryptography. ed25519-dalek [library]. https://github.com/dalek-cryptography/ed25519-dalek [Online; accessed 01/06/2021].

[45] dalek cryptography. merlin [library]. https://github.com/dalek-cryptography/merlin [Online; accessed 10/15/2020].

[46] Henry de Valance, Isis Lovecruft, and Tony Arcieri. Hash-to-Group with Elligator. https://ristretto.group/formulas/elligator.html [Online; accessed 10/08/2020].

[47] Henry de Valance, Isis Lovecruft, and Tony Arcieri. Ristretto. https://ristretto.group/ristretto.html [Online; accessed 10/05/2020].

[48] Henry de Valance, Isis Lovecruft, and Tony Arcieri. Ristretto in Detail: Equality Testing. https://ristretto.group/details/equality.html [Online; accessed 10/06/2020].

[49] Henry de Valance, Isis Lovecruft, and Tony Arcieri. What is Ristretto? https://ristretto.group/what_is_ristretto.html [Online; accessed 10/06/2020].

[50] Henry de Valance, Isis Lovecruft, and Tony Arcieri. Why Ristretto? https://ristretto.group/why_ristretto.html [Online; accessed 10/03/2020].

[51] Henry de Valence. Merlin Transcripts. https://merlin.cool/index.html [Online; accessed 10/15/2020].

[52] Henry de Valence, Jack Grigg, George Tankersley, Filippo Valsorda, Isis Lovecruft, and Mike Hamburg. The ristretto255 and decaf448 Groups. Internet-Draft draft-irtf-cfrg-ristretto255-decaf448-00, Internet Engineering Task Force, October 2020. Work in Progress.

[53] debian wiki. Microcode. https://wiki.debian.org/Microcode [Online; accessed 11/10/2020].

[54] debian wiki. ReproducibleBuilds About. https://wiki.debian.org/ReproducibleBuilds/About [Online; accessed 10/29/2020].

[55] Google Developers. Protocol Buffers Encoding. https://developers.google.com/protocol-buffers/docs/encoding [Online; accessed 01/07/2021].

[56] RustCrypto Developers. RustCrypto: BLAKE2 [library]. https://docs.rs/crate/blake2/0.9.0 [Online; accessed 10/06/2020].

[57] Whitfield Diffie and Martin 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].

[58] Changyu Dong. Math in Network Security: A Crash Course; Discrete Logarithm Problem. https://www.doc.ic.ac.uk/~mrh/330tutor/ch06s02.html [Online; accessed 11/26/2020].

[59] 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].

[60] eranrund. mc-watcher keeps log of attestation verification reports, Pull Request #694, February 2021. https://github.com/mobilecoinfoundation/mobilecoin/pull/694 [Online; accessed 02/11/2021].

[61] Alan Evans. slip10-ed25519-rust-crate [library]. https://gitlab.com/westonian/slip10-ed25519-rust-crate/-/tree/master [Online; accessed 04/05/2021].

[62] Giulia C. Fanti, Shaileshh Bojja Venkatakrishnan, Surya Bakshi, Bradley Denby, Shruti Bhargava, Andrew Miller, and Pramod Viswanath. Dandelion++: Lightweight cryptocurrency networking with formal anonymity guarantees. *CoRR*, abs/1805.11060, 2018. https://arxiv.org/pdf/1805.11060.pdf [Online; accessed 01/22/2021].

[63] 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].

[64] Vlad Filippov, Brian Warner, Artyom Pavlov, and RustCrypto Developers. HMAC-based Extract-and-Expand Key Derivation Function. https://lib.rs/crates/hkdf [Online; accessed 10/10/2020].

[65] 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].

[66] flawed.net.nz. Attacking Merkle Trees With a Second Preimage Attack, February 2018. https://flawed.net.nz/2018/02/21/attacking-merkle-trees-with-a-second-preimage-attack/ [Online; accessed 10/23/2020].

[67] 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].

[68] Forcepoint. What is Defense in Depth? https://www.forcepoint.com/cyber-edu/defense-depth [Online; accessed 10/23/2020].

[69] MobileCoin Foundation. fog [library]. https://github.com/mobilecoinfoundation/fog [Online; accessed 03/11/2021].

[70] MobileCoin Foundation. MobileCoin Fog Threat Model. https://github.com/mobilecoinfoundation/fog/blob/master/fog-threat-model-2.0.0.md [Online; accessed 05/25/2021].

[71] MobileCoin Foundation. mobilecoin [library]. https://github.com/mobilecoinfoundation/mobilecoin [Online; accessed 01/10/2021].

[72] MobileCoin Foundation. MobileCoin Main Net, December 2020. https://mobilecoinfoundation.medium.com/mobilecoin-main-net-8e355d82c726 [Online; accessed 12/30/2020].

[73] Jake Frankenfield. Simple Agreement for Future Tokens (SAFT), July 2020. https://www.investopedia.com/terms/s/simple-agreement-future-tokens-saft.asp [Online; accessed 12/30/2020].

[74] David Friedman. A Positive Account of Property Rights, 1994. http://www.daviddfriedman.com/Academic/Property/Property.html [Online; accessed 03/18/2020].

[75] 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].

[76] Steven Galbraith. Mathematics of Public Key Cryptography [v2.0 Chapter 21 extended version], 2012. https://www.math.auckland.ac.nz/~sgal018/crypto-book/ch21.pdf [Online; accessed 12/07/2020].

[77] Bob Glickstein. Understanding the Stellar Consensus Protocol, May 2019. https://medium.com/interstellar/understanding-the-stellar-consensus-protocol-423409aad32e [Online; accessed 01/26/2021].

[78] Joshua Goldbard. Notice of Exempt Offering of Securities [MobileCoin], May 2018. https://www.sec.gov/Archives/edgar/data/1739466/000173946618000001/xslFormDX01/primary_doc.xml [Online; accessed 12/30/2020].

[79] Joshua Goldbard and Moxie Marlinspike. MobileCoin, November 2017. https://mixin.one/assets/MobileCoin-Whitepaper-EN_FINAL.pdf [Online; accessed 04/07/2021].

[80] Shay Gueron. A Memory Encryption Engine Suitable for General Purpose Processors. Cryptology ePrint Archive, Report 2016/204, 2016. https://eprint.iacr.org/2016/204 [Online; accessed 12/01/2020].

[81] 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].

[82] Mike Hamburg. Decaf: Eliminating cofactors through point compression, 2015. https://www.shiftleft.org/papers/decaf/decaf.pdf [Online; accessed 10/05/2020].

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

[84] Huseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, and Ed Dawson. Twisted Edwards Curves Revisited. Cryptology ePrint Archive, Report 2008/522, 2008. https://eprint.iacr.org/2008/522 [Online; accessed 10/05/2020].

[85] imperva. Defense-in-Depth. https://www.imperva.com/learn/application-security/defense-in-depth/ [Online; accessed 10/23/2020].

[86] Intel. Attestation Service for Intel Software Guard Extensions (Intel SGX): API Documentation [Revision: 6.0]. https://api.trustedservices.intel.com/documents/sgx-attestation-api-spec.pdf [Online; accessed 12/01/2020].

[87] Intel. Intel Processors Load Value Injection Advisory. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00334.html [Online; accessed 02/09/2021].

[88] Intel. Intel Software Guard Extensions. https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html [Online; accessed 11/07/2020].

[89] Intel. linux-sgx [library]. https://github.com/intel/linux-sgx [Online; accessed 10/29/2020].

[90] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, May 2020. https://software.intel.com/content/dam/develop/public/us/en/documents/332831-sdm-vol-3d.pdf [Online; accessed 11/07/2020].

[91] Alon Jackson. Trust is in the Keys of the Beholder: Extending SGX Autonomy and Anonymity, May 2017. https://www.idc.ac.il/en/schools/cs/research/documents/jackson-msc-thesis.pdf [Online; accessed 12/01/2020].

[92] 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].

[93] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. Intel Software Guard Extensions: EPID Provisioning and Attestation Services. https://software.intel.com/content/dam/develop/public/us/en/documents/ww10-2016-sgx-provisioning-and-attestation-final.pdf [Online; accessed 12/01/2020].

[94] Simon Paul Johnson. An update on 3rd Party Attestation, December 2018. https://software.intel.com/content/www/us/en/develop/blogs/an-update-on-3rd-party-attestation.html [Online; accessed 12/01/2020].

[95] J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1, February 2003. https://tools.ietf.org/html/rfc3447 [Online; accessed 11/13/2020].

[96] 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].

[97] 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].

[98] Bradley Kjell. Big Endian and Little Endian. https://chortle.ccsu.edu/AssemblyTutorial/Chapter-15/ass15_3.html [Online; accessed 01/23/2020].

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

[100] koe, Kurt M. Alonso, and Sarang Noether. Zero to Monero — Second Edition, April 2020. https://web.getmonero.org/library/Zero-to-Monero-2-0-0.pdf [Online; accessed 10/03/2020].

[101] koe, James Cape, and Sara Drakeley. MobileCoin Key Derivation Upgrade: Migration to Mnemonics. https://medium.com/mobilecoin/mobilecoin-key-derivation-upgrade-migration-to-mnemonics-99a329a8b783 [Online; accessed 06/15/2021].

[102] koe and Sara Drakeley. MobileCoin Governance, Fees, and Supply. https://medium.com/mobilecoin/mobilecoin-governance-fees-and-supply-60c11782eb0a [Online; accessed 06/15/2021].

[103] H. Krawczyk, IBM, M. Bellare, UCSD, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication [RFC-2104], February 1997. https://tools.ietf.org/html/rfc2104 [Online; accessed 04/05/2021].

[104] H. Krawczyk, IBM Research, P. Eronen, and Nokia. HMAC-based Extract-and-Expand Key Derivation Function (HKDF) [RFC-5869], May 2010. https://tools.ietf.org/html/rfc5869 [Online; accessed 04/05/2021].

[105] 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].

[106] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem, 1982. https://lamport.azurewebsites.net/pubs/byz.pdf [Online; accessed 01/12/2021].

[107] 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].

[108] Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafał Malinowsky, and Jed McCaleb. Fast and Secure Global Payments with Stellar. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 80–96, New York, NY, USA, 2019. Association for Computing Machinery. https://www.scs.stanford.edu/~dm/home/papers/lokhava:stellar-core.pdf [Online; accessed 12/30/2020].

[109] Giuliano Losa, Eli Gafni, and David Mazières. Stellar Consensus by Instantiation, August 2019. https://www.scs.stanford.edu/~dm/home/papers/losa:stellar-instantiation.pdf [Online; accessed 05/05/2021].

[110] Ben Lynn. Lagrange's Theorem. https://crypto.stanford.edu/pbc/notes/group/lagrange.html [Online; accessed 12/02/2020].

[111] maciejhirsz. tiny-bip39 [library]. https://github.com/maciejhirsz/tiny-bip39 [Online; accessed 04/05/2021].

[112] 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].

[113] Gregory Maxwell. Confidential Transactions. https://elementsproject.org/features/confidential-transactions/investigation [Online; accessed 11/23/2020].

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

[115] David Mazières. The Stellar Consensus Protocol: A Federated Model for Internet-level Consensus, February 2016. https://www.stellar.org/papers/stellar-consensus-protocol?locale=en [Online; accessed 12/02/2020].

[116] David Mazières. Safety vs. Liveness in the Stellar Network, April 2019. https://www.scs.stanford.edu/~dm/blog/safety-vs-liveness.html [Online; accessed 03/08/2021].

[117] David Mazières, Giuliano Losa, and Eli Gafni. Simplified SCP, March 2019. http://www.scs.stanford.edu/~dm/blog/simplified-scp.html [Online; accessed 01/27/2021].

[118] 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].

[119] mfaulk. [MCC-1994] Adds validate_outputs_are_sorted, Pull Request #561, September 2020. https://github.com/mobilecoinfoundation/mobilecoin/pull/561 [Online; accessed 01/13/2021].

[120] mfaulk. Revert "[MCC-1969] Removes pseudo_outputs from range proof", Pull Request #565, November 2020. https://github.com/mobilecoinfoundation/mobilecoin/pull/565 [Online; accessed 01/13/2021].

[121] Daniel Miessler. What's the Difference Between a URI and a URL?, August 2020. https://danielmiessler.com/study/difference-between-uri-url/ [Online; accessed 12/28/2020].

[122] 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.

[123] 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].

[124] 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].

[125] 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].

[126] John C. Mitchell and Joe Zimmerman. Data-Oblivious Data Structures. 31st Symposium on Theoretical Aspects of Computer Science (STACS'14), 2014. https://core.ac.uk/download/pdf/62918559.pdf [Online; accessed 12/21/2020].

[127] MobileCoin. McCryptoBox. https://github.com/mobilecoinfoundation/mobilecoin/blob/master/crypto/box/README.md [Online; accessed 12/21/2020].

[128] MobileCoin. MobileCoin Consensus Service. https://github.com/mobilecoinfoundation/mobilecoin/blob/master/consensus/service/README.md [Online; accessed 02/01/2021].

[129] MobileCoin. RFC: Fog without a User Table, 2020. Unpublished manuscript.

[130] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008. http://bitcoin.org/bitcoin.pdf [Online; accessed 03/04/2020].

[131] Svetlin Nakov. ECIES Hybrid Encryption Scheme, November 2018. https://cryptobook.nakov.com/asymmetric-key-ciphers/ecies-public-key-encryption [Online; accessed 12/21/2020].

[132] 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].

[133] NIST. Side-Channel Attack. https://csrc.nist.gov/glossary/term/Side_Channel_Attack [Online; accessed 12/21/2020].

[134] Sarang Noether. Janus mitigation, Issue #62, January 2020. https://github.com/monero-project/research-lab/issues/62 [Online; accessed 02/17/2020].

[135] 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].

[136] 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].

[137] MobileCoin Official. full-service [library]. https://github.com/mobilecoinofficial/full-service [Online; accessed 03/10/2021].

[138] MobileCoin Official. mc-oblivious [library]. https://github.com/mobilecoinofficial/mc-oblivious [Online; accessed 03/09/2021].

[139] 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].

[140] 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].

[141] Trevor Perrin. The Noise Protocol Framework, July 2018. https://noiseprotocol.org/noise.pdf [Online; accessed 11/06/2020].

[142] Shaan Ray. Merkle Trees, December 2017. https://hackernoon.com/merkle-trees-181cb4bc30b4 [Online; accessed 10/20/2020].

[143] 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].

[144] Neptune Research and Isthmus. Monero tx_extra statistics, September 2020. https://github.com/neptuneresearch/monero-tx-extra-statistics-report [Online; accessed 10/13/2020].

[145] 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].

[146] SafeCurves. SafeCurves: choosing safe curves for elliptic-curve cryptography, 2013. https://safecurves.cr.yp.to/rigid.html [Online; accessed 03/25/2020].

[147] satoshilabs. Registered coin types for BIP-0044 [SLIP: 0044]. https://github.com/satoshilabs/slips/blob/master/slip-0044.md [Online; accessed 04/05/2021].

[148] satoshilabs. Universal private key derivation from master private key [SLIP: 0010]. https://github.com/satoshilabs/slips/blob/master/slip-0010.md [Online; accessed 04/05/2021].

[149] Vinnie Scarlata, Simon Johnson, James Beaney, and Piotr Zmijewski. Supporting Third Party Attestation for Intel SGX with Intel Data Center Attestation Primitives. https://software.intel.com/content/dam/develop/external/us/en/documents/intel-sgx-support-for-third-party-attestation-801017.pdf [Online; accessed 12/01/2020].

[150] Claus-Peter 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].

[151] 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].

[152] U.S. SEC. SEC Halts Alleged $1.7 Billion Unregistered Digital Token Offering, October 2019. https://www.sec.gov/news/press-release/2019-212 [Online; accessed 12/30/2020].

[153] U.S. SEC. SEC Charges Ripple and Two Executives with Conducting $1.3 Billion Unregistered Securities Offering, December 2020. https://www.sec.gov/news/press-release/2020-338 [Online; accessed 12/30/2020].

[154] Bassam El Khoury Seguias. Monero Building Blocks, 2018. https://delfr.com/category/monero/ [Online; accessed 10/28/2018].

[155] 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].

[156] Victor Shoup. *A Computational Introduction to Number Theory and Algebra*, pages 15–16. Cambridge University Press, 2005.

[157] SSLab. SGX 101. https://sgx101.gitbook.io/sgx101/ [Online; accessed 11/07/2020].

[158] Stellar. stellar-core [library]. https://github.com/stellar/stellar-core [Online; accessed 03/04/2021].

[159] sugargoat. Fee key env, Pull Request #438, September 2020. https://github.com/mobilecoinfoundation/mobilecoin/pull/438 [Online; accessed 12/31/2020].

[160] Andrew V. Sutherland. Schoof's algorithm, March 2015. https://math.mit.edu/classes/18.783/2015/LectureNotes9.pdf [Online; accessed 12/02/2020].

[161] Yogesh Swami. Intel SGX Remote Attestation is not sufficient, July 2017. https://www.blackhat.com/docs/us-17/thursday/us-17-Swami-SGX-Remote-Attestation-Is-Not-Sufficient-wp.pdf [Online; accessed 12/01/2020].

[162] 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].

[163] thomasrutter. What exactly is microcode and how does it differ from firmware? [thomasrutter's response], January 2018. https://superuser.com/questions/1283788/what-exactly-is-microcode-and-how-does-it-differ-from-firmware [Online; accessed 11/10/2020].

[164] Martin Thomson, Daniel Gillmore, and Benjamin Kaduk. Using Context Labels for Domain Separation of Cryptographic Objects, May 2016. http://www.watersprings.org/pub/id/draft-thomson-saag-context-labels-00.html [Online; accessed 10/08/2020].

[165] Certificate Transparency. How Log Proofs Work. http://www.certificate-transparency.org/log-proofs-work [Online; accessed 10/20/2020].

[166] UkoeHB. Proposal/Request: Update Supplementary Transaction Content, Issue #6456, April 2020. https://github.com/monero-project/monero/issues/6456 [Online; accessed 10/11/2020].

[167] UkoeHB. Reduce minimum fee variability, Issue #70, February 2020. https://github.com/monero-project/research-lab/issues/70 [Online; accessed 03/20/2020].

[168] UkoeHB. Reduce scan times with 1-byte-per-output 'view tag', Issue #73, April 2020. https://github.com/monero-project/research-lab/issues/73 [Online; accessed 11/23/2020].

[169] UkoeHB. RFC - MobileCoin Scheduled Hardforks with SCP and Dual Blocks, 2021. Unpublished manuscript.

[170] Nicolas van Saberhagen. CryptoNote V2.0. https://bytecoin.org/old/whitepaper.pdf [Online; accessed 03/10/2021].

[171] Adam "waxwing" Gibson. From Zero (Knowledge) To Bulletproofs, June 2020. https://github.com/AdamISZ/from0k2bp/blob/master/from0k2bp.pdf [Online; accessed 01/13/2021].

[172] 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].

[173] Wikibooks. Undergraduate Mathematics/Coset, 2020. https://en.wikibooks.org/wiki/Undergraduate_Mathematics/Coset [Online; accessed 10/06/2020].

Appendices

APPENDIX A

TXTYPE_RCT_1 Transaction Structure

We present in this appendix a dump from a MobileCoin transaction of type `TXTYPE_RCT_1`, together with explanatory notes for relevant fields. The dump was obtained from MobileCoin's 'full-service' wallet back-end [137] using the `get_mc_protocol_transaction` endpoint.

For editorial reasons, we have cut out large parts of the transaction, replacing them with ellipses and noting what was removed. We have also shortened long hexadecimal chains, presenting only the beginning and end as in `262a6dc1b4712123e1e06740fa8c2342f0c6[...]41d3a5286f`.

Once verified by nodes, all parts of a transaction, aside from components `outputs`, `fee`, and `key_image`, are discarded. Assuming transaction authors don't retain copies, and there are no SGX breaches being exploited, the contents of transactions 'merged' into the blockchain will be completely lost.

Our sample transaction has 2 inputs and 2 outputs.

```
"prefix": {
  "inputs": [
    {
      "ring": [
        {
          "amount": {
            "commitment": "262a6dc1b4712123e1e06740fa8c2342f0c6[...]41d3a5286f",
            "masked_value": "5812746746486261999"
          },
```

```
          "target_key": "9a2c70622ea9268d8bef3995cf0e8b55ca64bd[...]bf6e264f19",
          "public_key": "105bd961094980a985e655f525c3d4eb519f89[...]e6e88caa21",
          "e_fog_hint": "be9c4fb3e1ae4dd3ac974cf9b912c8ec0d6fbd[...]90ac970100"
        },
        {
          "amount": {
            "commitment": "38cb7ac811d299d78ca688d5320c7839bb05[...]831e27d162",
            "masked_value": "8485203080731247802"
          },
          "target_key": "4ce5165723f3ac35777ad05b92dc3d1673790c[...]40ed7d5827",
          "public_key": "1402da2d31deff93fc4aad7d44ba5f010bce06[...]c7f4018d65",
          "e_fog_hint": "000000000000000000000000000000000000[...]0000000000"
        },
        ... 9 more ring members
      ],
      "proofs": [
        {
          "index": "1367",
          "highest_index": "12971",
          "elements": [
            {
              "range": {
                "from": "1367",
                "to": "1367"
              },
              "hash": "7f14b20b05d3cbc8a2d7695aecee5fd41df9a51a[...]e27402d7cf"
            },
            ... 14 more elements
          ]
        },
        ... 10 more membership proofs
      ]
    }
  ],
  "outputs": [
    {
      "amount": {
        "commitment": "eabec03023e61b94be61b3d5a0d7b692a17fbc8b[...]c1f8bf0b67",
        "masked_value": "15930548794624239600"
      },
      "target_key": "3ef6413f649c1153d9f727bff7e2b8e17e5df6c9de[...]f90f7f1b3d",
      "public_key": "6acd0c9400aafaf7bfcec6649b5527638eeb3ebaca[...]ad3ebb3c05",
```

```
      "e_fog_hint": "1322b802c2ecbd392c726686818a5d4023f95f343c[...]a496cdfafb"
    },
    {
      "amount": {
        "commitment": "781cc27424252c9f16657dc2935e3ab3972362af[...]4508d38db0",
        "masked_value": "17147565098074236509"
      },
      "target_key": "3643866e770086dd46e5f7728c887bde4d52fbf866[...]8d3dae341e",
      "public_key": "70142cf2cf81433c2d0f034e95af8d214e11f98a83[...]e52a7ff5f1",
      "e_fog_hint": "5bfdcd4be7d3efc0a85e133dee0a9273fa7c6ef521[...]04e9e10101"
    }
  ],
  "fee": "10000000000",
  "tombstone_block": "4377"
},
"signature": {
  "ring_signatures": [
    {
      "c_zero": "5422f19a19f1ae8fae392970f1956917370326fd4d8b50[...]a9b5238fe7",
      "responses": [
        "6ffae81179a2f7d08b80ff2e65f989afa7ac4c2aad269b565f1d5d8905fb3f0a",
        "235c6e815af40a0bb771b4523421eb070d7914c3c98426be7864aaee6b929e0a",
        "cdd77b59b391c0f23e3d536e413c8e2148876a9550efd641ae71811693bc440e",
        "cb03858631a62681b5a9e6bcb31d9dc057a137e71d00eeecbbc1a2b035d33c0c",
        "f95120a9e8790b835a9b07831577841c516a4b02bba3fd7043a0243411208c0f",
        "90bf683446463a3bfdcb22d4f8761b370347ad6839dea64a428da274e872b50f",
        "5243672def3f1550a5e4835c3b32b6716fcc8bd2d0cdfdf58eae1bea2fc38409",
        "0d5e5b466d93d99b14312c582de9a70f0f517ab391457892dd31f411dba5de01",
        "97e0b088227650437417408417 06b2ebf943cbb7155ae1da861875c0f9683000",
        "f64b00670f337646e59adfe7d9445bfe3a2a7bdb3c6979962dc79b531faf400c",
        "7a1f4d9110783a03e2086178881a605ca3c200741ad1d0895d68dcc0c3f8040f",
        "2be0531a2d377ef3b61de7990cc7738dcd7204b7c334f811586df87f659c8705",
        "8dd611f6fa1e7aa37ad31eec622651ec2aa6fac025ce54dcdbf35b257b441c09",
        "0caac06183b9a6e25ec56aa7b5dc32d38f3258fcdf5033bfbfd5fc1d8573b304",
        "8d60665d8e45f46bef44a107bfcb1b5025d25ba8480d3e0480c85b0cc2644507",
        "4083f83e9032e185b64177613689cbcab8e3b552ff2fc4b698bb8dda6bca6d0d",
        "9c5e7695ae7997f9135d297246053d3198d74f6696f9494b66e8137b27c2dd0a",
        "74c4fee37278af3f65043856b0d50856b0501a4ec15ced28d5fd63b77e97b10d",
        "3a3b948b7b9bc4a209908c4c12bad195a91d9e600977767993b7ef6db97de305",
        "5403ca939dda2c5fc9ed22eadc3c858e387bb82dc244ab29b1d448d94e639d04",
        "c4daf4e968b3e69c50c50061596e46db481db6989faec2fa8ea51061b7f96e00",
        "9b0d053337ae82fc4e4d6904c9942f8aec28a03502b101a48e2220adbf306206"
```

```
94        ],
95        "key_image": "c2cf8a58a9f1f5fcdd88181d16eede697f176980b6b[...]dee3dc79a3"
96      }
97    ],
98    "pseudo_output_commitments": [
99      "6e566450428b4a589a7f97fe88d60a558b327ab56ae03076494dbb658675f00a"
100   ],
101   "range_proofs": "ba5e39a27daedc0f02a9bd19f9b83d8394ab6d[...]8f6ef8745e53a009"
102 }
```

Transaction components

- `inputs` (lines 2-43) - List of inputs (there is one here).
- `ring` (lines 4-24) - Ring members of the first input. These are outputs copied explicitly from the blockchain record into the transaction structure.
- `commitment` (line 7) - Commitment for the first ring member of the first input (Section 5.4).
- `masked_value` (line 8) - Field $masked\_value_t$, for the first ring member of the first input (Section 5.3).
- `target_key` (line 10) - One-time destination key (address) for the first ring member of the first input (Section 4.2).
- `public_key` (line 11) - The transaction output (txout) public key for the first ring member of the first input (Section 4.2).
- `e_fog_hint` (line 12) - Encrypted fog hint for the first ring member of the first input (Section 11.2.3). Note that the second ring member's fog hint in line 21 is all zeroes, implying it is a fee output. Currently MobileCoin fee outputs are not differentiated from normal outputs when constructing transactions, even though practically speaking fee outputs (which are public knowledge — recall Section 9.4 footnote 17) are not useful as decoy members of a ring signature. Since MobileCoin blocks are created very quickly once nodes start working on an SCP slot, most blocks will only 'contain' one transaction unless MobileCoin transaction volume becomes very large. Combine this with the fact most transactions will have two outputs (one for the recipient, one for sending change back to the sender) means on average one-third of on-chain outputs will be fee outputs. Therefore the average effective ring size is two-thirds its nominal value (11 in the initial version of MobileCoin, so about 7-8 on average). One way MobileCoin could reduce this problem (which also leads to unnecessary blockchain bloat) is implementing a 'fee epoch' where fees are recorded in clear text in blocks, and nodes only create a fee output e.g. every 1000 blocks.
- `proofs` (lines 25-41) - Membership proofs showing the first ring's members exist in the blockchain (Section 6.2.2).

- `index` (line 27) - Index of the first ring member of the first input (i.e. the output's index within the blockchain record).
- `highest_index` (line 28) - Index of the newest output in the blockchain at the time this membership proof was created. Technically it could be any existing output in the blockchain with an index higher than the ring member, but in practice it is likely to be the newest output (at the time the proof was created).
- `elements` (lines 29-38) - Membership proof elements, each is a node in the proof's Merkle tree.
- `range` (lines 31-34) - Range of elements this proof node represents. Since the `from` and `to` fields are the same, this must be a leaf node.
- `hash` (line 25) - The hash value of this proof node.
- `outputs` (lines 44-63) - The outputs created by this transaction (there are two). Outputs have the same content fields as ring members, since ring members are just outputs copied from the blockchain.
- `fee` (line 64) - Transaction fee in clear text, in this case 0.01 MOB (Section 9.4).
- `tombstone_block` (line 65) - A future block's index selected by the transaction author. The transaction must be added to the blockchain before this block is created, otherwise it is considered 'dead' and is no longer eligible for adding to the chain.
- `ring_signatures` (lines 68-96) - MLSAG signature for the transaction input (there is one; Section 7.2.2).
- `c_zero` (line 70) - Component c_1 from the first input's MLSAG signature (Section 3.5; note that the MobileCoin implementation uses 0-base indexing for MLSAGs, but for consistency/clarity we stick with 1-base indexing).
- `responses` (lines 71-94) - In alternating order, components $r_{i,1}$ and $r_{i,2}$ from the first input's MLSAG signature

$$\sigma_j(\mathfrak{m}) = (c_1, r_{1,1}, r_{1,2}, ..., r_{v+1,1}, r_{v+1,2})$$

- `key_image` (line 95) - Key image $\tilde{K}_j$ for the first input ($j = 1$; Section 3.4).
- `pseudo_output_commitments` (lines 98-100) - Pseudo output commitment $C_j'^a$ for the input (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).
- `range_proofs` (line 101) - Bulletproof proof elements concatenated together into one blob (Bulletproofs were not explored in this document, so we will not itemize further):

$$\Pi_{BP} = (A, S, T_1, T_2, t_x, t_x^{blinding}, e^{blinding}, \mathbb{L}, \mathbb{R}, a, b)$$

APPENDIX B

Block Content

In this appendix, we show the structure of a sample block, namely the 5^{th} block after the origin block (i.e. the 6^{th} block overall), obtained from the MobileCoin 'full-service' wallet back-end [137] with endpoint `get_block`. The block has two outputs and one key image.

We can infer this block was created from one transaction with one input and one output, since all transactions must have an input (but there is only one key image here), and one output from every block must be for the block fee.


```
"block": {
  "id": "3f30b6d56cd5045419d7607517ccc6cbea62839d772afe8642d8075072cf7f11",
  "version": "0",
  "parent_id": "46504c990b572eb48a7cd29cb5c8891b90513678a290e6e7235ab8947644182a",
  "index": "5",
  "cumulative_txo_count": "28",
  "root_element": {
    "range": {
      "from": "0",
      "to": "31"
    },
    "hash": "72276f5c57bcfc585050c6b4f33eefcc6f0a47ef79aeeaa68593af608108b7e9"
  },
  "contents_hash": "f802d38eb533a8497099f8e04564943abdc6e34c566c2045bbeaa46e0ad170b4"
},
```

```
16 "block_contents": {
17   "key_images": [
18     "0a204451a2b6c0795f1814f5cf490d8b69c9fb9c244252d88b76be842ab1ec4ca22e"
19   ],
20   "outputs": [
21     {
22       "amount": {
23         "commitment": "cafbcf8f05b29a7f231cbb4c955f3cd8997b32d50758ceda2de3591c314d083f",
24         "masked_value": "12747939974205126347"
25       },
26       "target_key": "58a5cd624cb0eaf44c5429e8e9dd53ca339ee21819847fe0f09307c1bd7a6e3e",
27       "public_key": "86347301b9910e067fc94ac9c90381fc8dd521812006af48440fbd9f64eeea20",
28       "e_fog_hint": "00000000000000000000000000000000000000000[...]00000000000000000000"
29     },
30     {
31       "amount": {
32         "commitment": "1640896f882ea29d0bf6de88cc30b59fa520509c3905e5a8a55cefdde7e52f55",
33         "masked_value": "15060832203020249l9"
34       },
35       "target_key": "a82316cd7b844689ca7d907cda38fb48a660c301aaee3a277791a11a34e56772",
36       "public_key": "f451c9927e6b1c73265f37bc93b7a042addc627137ab1082d8f42b88b2917a6f",
37       "e_fog_hint": "f1b2f8a048385516a936affc895e0356120f15[...]765c36adaae2981910100"
38     }
39   ]
40 }
```

Block components

- `block` (lines 1-15) - This block's header.
- `id` (line 2) - This block's block ID.
- `version` (line 3) - Block format version.
- `parent_id` (line 4) - Parent block's ID.
- `index` (line 5) - This block's index in the ledger.
- `cumulative_txo_count` (line 6) - Total number of outputs in the ledger (including those added by this block).
- `root_element` (lines 7-13) - Merkle root of the outputs that existed in the chain before this block was created (recall Chapter 6).

- `range` (lines 8-11) - Range of elements covered by the root element (i.e. the size of the Merkle tree's base layer). Since there were only 26 outputs on-chain before this block was made, there must be 6 null elements in the base layer of the tree that made this root element.
- `hash` (line 12) - The Merkle root hash of the root element.
- `contents_hash` (line 14) - A hash of the block contents.
- `block_contents` (lines 16-40) - All block contents added to the chain by this block.
- `key_images` (lines 17-19) - Key images added to the chain by this block, corresponding to outputs spent by the transactions validator enclaves used to assemble this block. There is only one key image here.
- `outputs` (lines 20-39) - New outputs added to the chain by this block.
- `commitment` (line 23) - The first output's commitment (recall Chapter 5).
- `masked_value` (line 24) - The amount contained in the output commitment, masked so only the recipient can read it (recall Section 5.3).
- `target_key` (line 26) - One-time address of the output.
- `public_key` (line 27) - Txout public key of the output.
- `e_fog_hint` (line 28) - Encrypted fog hint of the output. The first output's fog hint is all zeroes, likely indicating it is the fee output of this block.

APPENDIX C

Origin Block

In this appendix, we show the structure of MobileCoin's origin block, obtained from the MobileCoin 'full-service' wallet back-end [137] with endpoint `get_block`. The block has 16 outputs and no key images. Those outputs contain the entire original supply of MobileCoin (250 million MOB) [102].

To verify that those 16 outputs contain the total supply (15.625 million MOB each), restore the 'Origin Account' from its root entropy (see [102]). Root entropies are deprecated as a mechanism for creating MobileCoin accounts [101], but for backward compatibility are still supported by the 'full-service' wallet back-end [137] with endpoint `import_account_from_legacy_root_entropy`.

- Root entropy: `aa9082086d4f341300d3ba08d41db566edc9fb61eec623870b80871681a589eb`
- View private key: `d825a5c3aaef812d17feba5cf8f3019bcb0608c0a673200fa425e56eb1dc2d0b`
- Spend private key: `9e95df3801c4b6ab0be854377430e18528546be71335e2afb91f058b3013380f`

The endpoint `get_all_txos_for_account` will display all outputs the account has received. Outputs with `received_block_index = 0` were found in the origin block.

The contents of this block are fairly self-explanatory. See Appendix B for an overview of block contents.


```
"block": {
  "id": "c502dcac6985ade2766af6a4bd1973f4af71aa4136c6f6bf61001b21e5ddf9ae",
  "version": "0",
  "parent_id": "0000000000000000000000000000000000000000000000000000000000000000",
```

```
  "index": "0",
  "cumulative_txo_count": "16",
  "root_element": {
    "range": {
      "from": "0",
      "to": "0"
    },
    "hash": "0000000000000000000000000000000000000000000000000000000000000000"
  },
  "contents_hash": "a099ec3bfedc4a9c5e876a3ef1a35a972e7789a3cd35a4b842094c517d631531"
},
"block_contents": {
  "key_images": [],
  "outputs": [
    {
      "amount": {
        "commitment": "9e6f1564d04a6e60a3d989dba26426b07e0ef9cc1e63915e538fd06ba56b0558",
        "masked_value": "16611767762345837"
      },
      "target_key": "aaa06df144812f7c09e41e3414fa5dedfa8628e41981880c36d13c5e58cac11e",
      "public_key": "bcdf3c85183307bcecb0f99734118f29f5bc74e79eae44bd577606de9f129405",
      "e_fog_hint": "436f7669642056616363696e6520557020746f20393[...]3066633431373535"
    },
    {
      "amount": {
        "commitment": "6aebc04b8d5a901064f2f5306aebab723184c22041e8b6552310825c8857ed14",
        "masked_value": "13677070265391552919"
      },
      "target_key": "1a3803da55c57e8016a040bda881e96b8beb203f631df57562ac873b24eec301",
      "public_key": "0a219aad7e9066bc317b05f57d4555fc21ea58d61255c3e128b61cae5ed4545f",
      "e_fog_hint": "436f7669642056616363696e6520557020746f20393[...]3066633431373535"
    },
    ... 14 more outputs
  ]
}
```