Mechanics of MobileCoin: First Edition

exploring the foundations of a private digital currency

February 5, 2021, Preview (8/11) v0.0.30

koe[1,2]

[1] ukoe@protonmail.com
[2] Author 'koe' worked on this document as part of a private contract with MobileCoin, Inc.

Abstract

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

It is common knowledge that cryptography is used to secure communications, whether they be coded letters or private digital interactions. Another application is in so-called cryptocurrencies. These digital moneys use cryptography to assign and transfer ownership of funds. To ensure that no piece of money can be duplicated or created at will, cryptocurrencies usually rely on 'blockchains', which are public, distributed ledgers containing records of currency transactions that can be verified by third parties [93].

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 [120]. 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 [93], where blocks are consensuated with an implementation of the Stellar Consensus Protocol [85], transactions are validated in SGX secure enclaves [31] and are based on elliptic curve cryptography using the Ristretto abstraction [56] on curve Ed25519 [23], transaction inputs are shown to exist in the blockchain with Merkle proofs of membership [86] and are signed with Schnorr-style multilayered linkable spontaneous anonymous group signatures (MLSAG) [97], and output amounts (communicated to recipients via ECDH [43]) are concealed with Pedersen commitments [83] and proven in a legitimate range with Bulletproofs [28].

Contents

CHAPTER 1

Introduction

In the digital realm it is often trivial to make endless copies of information, with equally endless alterations. For a currency to exist digitally and be widely adopted, its users must believe its supply is strictly limited. A money recipient must trust they are not receiving counterfeit coins, or coins that have already been sent to someone else. To accomplish 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." [12] 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' [91], 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 [102].

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 [94]. However, with appropriate tools it is possible to analyze flows and to a large extent link true senders with receivers [110, 25, 98, 29].

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 version 1 of the MobileCoin protocol[5], corresponding to version 1.0.1 of the MobileCoin software suite. All transaction and blockchain-related mechanisms described here belong to those versions.[6]

[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: [48].

[4] Seguias has created the excellent Monero Building Blocks series [109], 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] The MobileCoin codebase's integrity and reliability is predicated on assuming enough people have reviewed

1.2 Readership

We anticipate many readers will encounter this report with little to no understanding of discrete mathematics, algebraic structures, cryptography[7], 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.[8]

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." [54]

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.[9]

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.

it to catch most or all significant errors. We hope that readers will not take our explanations for granted, and verify for themselves the code does what it's supposed to. If it 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.

[7] An extensive textbook on applied cryptography can be found here: [26].

[8] 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.

[9] Monero, initially known as BitMonero, was created in April 2014 as a derivative of the proof-of-concept currency CryptoNote [115]. 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.

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.[10] 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?

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

- [dalek25519]: the `dalek-cryptography curve25519-dalek` library [34]

[10] 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.

- [dalekEd]: the `dalek-cryptography ed25519-dalek` library [36]
- [dalekBP]: the `dalek-cryptography bulletproofs` library [35]
- [blake2]: the Rust implementation [42] of hashing algorithm Blake2 [18]

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 an internal code base at MobileCoin (which has not yet been made open source) is tagged with [MC-prop]. 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.[11] 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 [79]. 'Zero to Monero' itself (its first edition was published in June 2018 [17]) is an expansion of Kurt Alonso's master's thesis, 'Monero - Privacy in the Blockchain' [16], 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.
- 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).

[11] 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).

- MobileCoin's consensus protocol (an implementation of the Stellar Consensus Protocol [85]) replaced Monero's more standard mining-based (Nakamoto [93]) 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 [16], 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.

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}. [77]

[2] The BLAKE2 hashing algorithm is a successor to the NIST standard *SHA-3* [13] finalist BLAKE. The BLAKE2b variant is optimized for 64-bit platforms. [19]

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

$$1 * 2^0 * 8 + 1 * 2^1 * 8 + 1 * 2^2 * 8 + 0 * 2^3 * 8$$
$$= 8 + 16 + 32 + 0 * 64$$

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] $\overset{?}{=}$ 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 $e_{scalar} \to e_{binary}$; $A = [e_{binary}]$; $r = 1$; $m = a$
2. Iterate through: $i = (0, ..., A_{size} - 1)$
   (a) If A[i] $\overset{?}{=}$ 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 [111].[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* [4]:

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

More generally (and more rapidly), the so-called 'extended Euclidean algorithm' [3] 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. It turns out $254 \pmod{13} \equiv 2*10*10+5*10+4 \equiv (((2) * 10 + 5) * 10 + 4) \pmod{13}$. An algorithm for $a \pmod n$ when $a > n$ is:

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

2.3 Elliptic curve cryptography

2.3.1 What are elliptic curves?

A finite field $\mathbb{F}_q$, where q is a prime number greater than 3, is the field formed by the set $\{0, 1, 2, ..., q-1\}$. Addition and multiplication $(+, \cdot)$ and negation $(-)$ are calculated $\pmod q$.

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

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

Typically, an elliptic curve is defined as the set of all points with coordinates (x, y) satisfying a *Weierstraß* equation [63] (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 [22], 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 [24] 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, $(x, y) \to (-x, y)$ [22], and use normal point addition. Recall that ‘negative’ elements $-x$ of $\mathbb{F}_q$ are really $-x \pmod q$.

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

[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 [64]) prior to curve operations like point addition, in order to avoid performing field inversions for efficiency. [122]

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.

$$\begin{aligned} I + P_2 &= P_2 \\ P_2 + P_2 &= P_4 \\ P_4 + P_2 &= P_6 = I \end{aligned}$$

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.

$$\begin{aligned} I + P_4 &= P_4 \\ P_4 + P_4 &= P_2 \\ P_2 + P_4 &= P_6 = I \end{aligned}$$

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 $n \pmod{u} \overset{?}{=} 0$ just multiply by u or return the 0th position directly (I). The orders of all possible subgroups are divisors of N (by *Lagrange's theorem* [81]).

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

1. Find N (e.g. use *Schoof's algorithm* [113]).
2. Find all the divisors of N.
3. For every divisor n of N, compute nP.
4. The smallest n such that $nP \overset{?}{=} I$ 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. [22].

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. [44]

[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* [43] 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 [107], generalized by Maurer in 2009 [82], that allows someone to prove they know the private key k of a given public key K without revealing any information about it [84]. 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 $R \overset{?}{=} R'$.

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. [53]

[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 $(\text{mod } l)$. 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 [108] 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.* [84]

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 [84].[18]

Using a hash function, instead of the verifier, to generate challenges is known as a *Fiat-Shamir transform* [46], because it makes an interactive proof non-interactive and publicly verifiable [84].[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, $\mathcal{H}(A) \in^D_R \mathbb{S}_H$ where $\mathbb{S}_H$ is the set of possible outputs from $\mathcal{H}$. We use $\in^D_R$ 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' [117] 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, but unless otherwise stated all uses of hash function have their own tag.

[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 [71], ANSI X9.62, and [63] 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.* [22, 23, 24].

[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 [61]. 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 [9].

Curve Ed25519 is not subject to any patents (see [78] for a discussion on this subject), and the team behind it has developed and adapted basic cryptographic algorithms with efficiency in mind [24].

[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 [62].

For example, it is possible to add a point from the subgroup of size h (the cofactor) to a point P, and, with a scalar n that is a multiple of h, create several points which when multiplied by n have the same resultant point. This is because an EC point multiplied by its subgroup's order is 'zero'.[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. [105]

[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 $l\tilde{K} \overset{?}{=} 0$ [50].

The importance of using prime-order-subgroup points was the motivation behind Ristretto, which is an encoding abstraction for twisted Edwards-based curves that efficiently constructs a prime-order group out of the underlying non-prime group. [59]

2.4.2 Ristretto

Ristretto can be thought of as a 'binning' procedure for twisted Edwards curve points. 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.

The important innovation of Ristretto is each bin has a representative 'canonical member' that 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 $K + K_c^h$ (where c may equal a or b, or be a different cofactor point).

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' [123].

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 [56], but we can give a brief insight into how it works. Normal point compression for the Ed25519 curve was standardized in [75], first described in [23], and the concept was introduced in [90].

As background, it's helpful to know 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 $\pmod{q}$, so there are no actual negative values. However, taking $\pmod{q}$ of $-x$ will change the value between odd and even since q is odd. For example: $3 \pmod{5} = 3$, $-3 \pmod{5} = 2$. In other words, the field elements x and $-x$ have different odd/even assignments.

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

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* [64] $(X : Y : Z : T)$, 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 and notes [56] for any production-level applications.

2. Compute[30] $z = uv^3(uv^7)^{(q-5)/8} \pmod q$.
   (a) If $vz^2 \overset{?}{=} u \pmod q$, set $r = z$.
   (b) If $vz^2 \overset{?}{=} -u \pmod q$, calculate $r = z * 2^{(q-1)/4} \pmod q$.
3. If the least significant bit of r is 1 (i.e. it is odd), return $-r$, otherwise return r.[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 $= \texttt{Sqrt}(1, u_1u_2^2) \pmod q$.
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 $b \overset{?}{=} 1$, 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 $b \overset{?}{=} 0$, 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

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

1. Given a supposed compressed curve point s, check if it is a valid field element with a byte-wise comparison $s \pmod q \overset{?}{=} s$. Reject s if it is odd or invalid.

[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 [23] as field elements with the least significant bit not set (i.e. 'even' field elements). In normal Ed25519 point decompression [23] 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.

2. Compute
$$y = \frac{1+as^2}{1-as^2} \pmod q$$
3. Compute (x should be 'even' after this step)
$$x = \texttt{Sqrt}(4s^2, ad(1+as^2)^2 - (1-as^2)^2) \pmod q$$
4. Convert to extended coordinates if desired.
$$\begin{aligned} X &= x \\ Y &= y \\ Z &= 1 \\ T &= xy \pmod q \end{aligned}$$

We can use extended coordinates to test if two points belong to the same Ristretto bin. If either $X_1Y_2 \overset{?}{=} Y_1X_2$ or $Y_1Y_2 \overset{?}{=} -aX_1X_2$ 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 [57].[32,33]

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

Implementations of Ed25519 typically use the generator $G = (x, 4/5)$ [23], 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 [10]. 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 $Y_1Y_2 \overset{?}{=} X_1X_2$.
[33] Multiple extended coordinates can represent a given curve point [58], 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 [19] into a 64-byte bit string, which is passed to `from_bytes_mod_order_wide()` and 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 [30]), 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 [80] building on the work of Rivest *et al.* in [104]. 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). [80]

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 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 $n+1 \rightarrow 1$,
$$c_{i+1} = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [r_i G + c_i K_i])$$
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 $n+1 \rightarrow 1$,
$$c'_{i+1} = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [r_i G + c_i K_i])$$
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 [80], 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 [97] based on a publication by Adam Back [21] regarding the CryptoNote [120] 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 [52].

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]

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

[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 [97] and [80].

[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 [55], 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>()`

[12] In MobileCoin it is important to use the hash to point function for key images instead of another base point so linearity doesn't lead to linking signatures created by the same address (even if for different one-time addresses). See [120] page 18.

2. Generate random number $\alpha \in_R \mathbb{Z}_l$ and random numbers $r_i \in_R \mathbb{Z}_l$ for $i \in \{1, 2, ..., n\}$ but excluding $i = \pi$.

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

4. For $i = \pi + 1, \pi + 2, ..., n, 1, 2, ..., \pi - 1$ calculate, replacing $n + 1 \rightarrow 1$,
$$c_{i+1} = \mathcal{H}_n(\mathfrak{m}, [r_i G + c_i K_i], [r_i \mathcal{H}_p(K_i) + c_i \tilde{K}])$$

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

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 [76] suggests it may not be necessary, although adding the prefix is standard practice for similar signature schemes (LSAG uses key prefixing).

[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 [50]. 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 [97], 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 [80].

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

[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]

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).[4]

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.[5] 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.[6]

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

Bob has private/public keys (k^v_B, k^s_B) and (K^v_B, K^s_B), and Alice knows his public keys (his address). The mock transaction could proceed as follows [120]:

[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 [14]. In brief, addresses are set in a 'protobuf' format [41], 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] In the core MobileCoin implementation, the view and spend private keys are derived from a so-called 'root entropy' with a 'key derivation function'. This boils down to a domain-separated hash of a random 32-byte number u, such that $k_v = H_{Blake2b256}(u, \text{“view”}) \pmod{l}$ and $k_s = H_{Blake2b256}(u, \text{“spend”}) \pmod{l}$. Oddly, deriving private keys doesn't use the typical $H_n()$ hash function mentioned in Section 3.1, but instead a wrapper around Blake2b that creates 32-byte outputs instead of 64-bytes. The hash wrapper is used by the RustCrypto library `KDFs` [47] to hash the inputs. A person only needs to save the root entropy u in order to access (view and spend) all of the outputs they own (spent and unspent).

account-keys/src/identity.rs `AccountKey::from()`

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

[6] Except with negligible probability.

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

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

[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.[8]

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 $K_B'^s \overset{?}{=} K_B^s$, 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.[9,10]

4.3 Subaddresses

MobileCoin users can generate subaddresses from each address [96]. Funds sent to a subaddress can be viewed and spent using its main address's view and spend keys. By analogy: an online

[7] 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.

[8] In CryptoNote the key rG is termed the 'transaction public key' [120], while MobileCoin re-brands it to 'transaction output public key' to better express its role.

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

[10] 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).

bank account may have multiple balances corresponding to credit cards and deposits, yet they are all accessible and spendable from the same point of view — the account holder.

Subaddresses are convenient for receiving funds to the same place when a user doesn't want to link his activities together by publishing/using the same address. As we will see, most observers would have to solve the DLP to determine a given subaddress is derived from any particular address [96].[11]

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()`**

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

So,

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

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.
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.[12]

[MC-tx] src/onetime_keys.rs **`recover_public_subaddress_spend_key()`** mobilecoind/src/database.rs **`get_subaddress_id_by_spk()`**

[11] 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.

[12] An advanced attacker may be able to link subaddresses [45] (a.k.a. the Janus attack). With subaddresses (one of which can be a normal address) K_B^1 & K_B^2 the attacker thinks may be related, he makes a transaction output with $K^o = \mathcal{H}_n(rK_B^{v,2})G + K_B^{s,1}$ 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 [119], the most efficient known mitigation is encrypting the txout private key r and adding it to transaction data. See [95] for background on this topic. We are not aware of any wallets that have implemented a mitigation for this attack. See Section **??** footnote **??** for a different mitigation that could be added to MobileCoin in the future.

Bob only needs his private view key k^v_B and subaddress public spend key $K^{s,1}_B$ to find transaction outputs sent to his subaddress.

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

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

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).[13]

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()`

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

Moreover, to promote uniformity the core MobileCoin implementation never passes users' normal addresses out for receiving funds. Instead, only subaddresses are made available.[14] 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 [120], the txout public key will likely never appear in the form r_tG in the blockchain, but will instead always be $r_tK^{s,i}_t$.

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 implemented by a third party. Future editions of 'Mechanics of MobileCoin' may discuss this topic, but for now we defer to the treatment in Chapters 9 and 10 of [79].

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

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

[14] 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.

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 $h = \mathcal{H}(\text{“My secret”}, \textit{heads})$ 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 $h' = \mathcal{H}(\text{“My secret”}, \textit{heads})$. 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* [99] 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 [83, 108].

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, $x \rightarrow xG$. 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' [121]. Even the committer can't find another combination without solving the DLP for γ, a property called 'computational binding' [121].

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 $r_t K^v_B$ 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]

$$y_t = \mathcal{H}_n(\text{“}mc\_amount\_blinding\text{”}, \mathcal{H}_n(r_t K^v_B))$$
$$masked\_value_t = b_t \oplus_8 \mathcal{H}_n(\text{“}mc\_amount\_value\text{”}, \mathcal{H}_n(r_t K^v_B))$$

[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^v_B. He can also check that the commitment $C(y_t, b_t)$ provided in the output data, henceforth denoted C^b_t, 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 **??**).

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 [97] 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 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 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

$$C^a_j = x_j G + a_j H$$

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

$$C'^a_j = x'_j G + a_j H$$

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

$$C^a_j - C'^a_j = (x_j - x'_j)G$$

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 $C^a_j - C'^a_j = z_j G + 0H$, which we will actually do in Chapter 7 when we discuss the structure of RingCT transactions.

Let us call C'^a_j 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:

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

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

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

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

[8] Recall from Section 2.3.1 we can subtract a point by inverting its coordinates then adding it. If $P = (x, y)$, $-P = (-x, y)$. Recall also that negations of field elements are calculated $\pmod q$, 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 [28] (and also explored in [121, 33]).[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^b_t and an n-tuple aggregate proof[10,11]

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

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

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^b_t, 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]

[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 2ˆ189 times bigger than the available range, which means a ridiculous 2ˆ189 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 [35] for verification.

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

[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 transaction volume is significant.

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

[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 [88], 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 [86].

6.1 Merkle trees

Merkle trees [86] are a simple and effective way of taking a large data set and representing it with a much smaller identifier [118]. They are well-known and see widespread use among blockchain-based technologies [101].

[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.

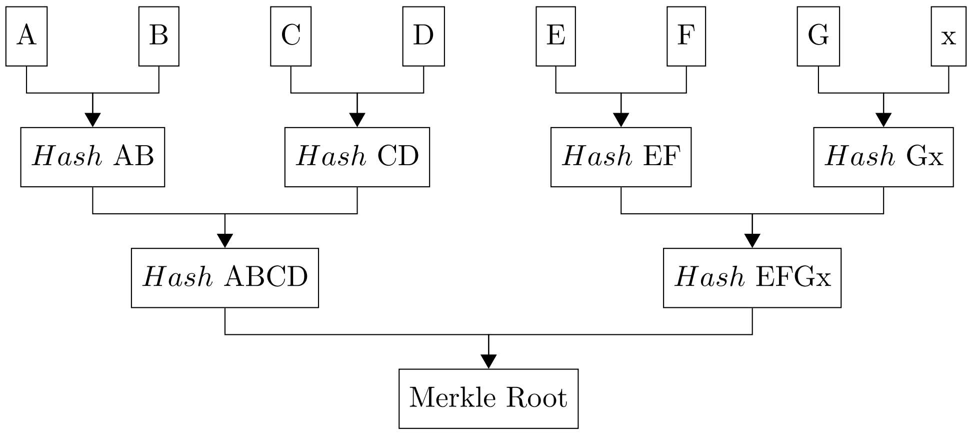

Diagram 6-1: Merkle Tree

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 [101, 118]. 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' [51, 65] against second preimage attacks [49] alongside the de facto defense of implementation robustness.

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

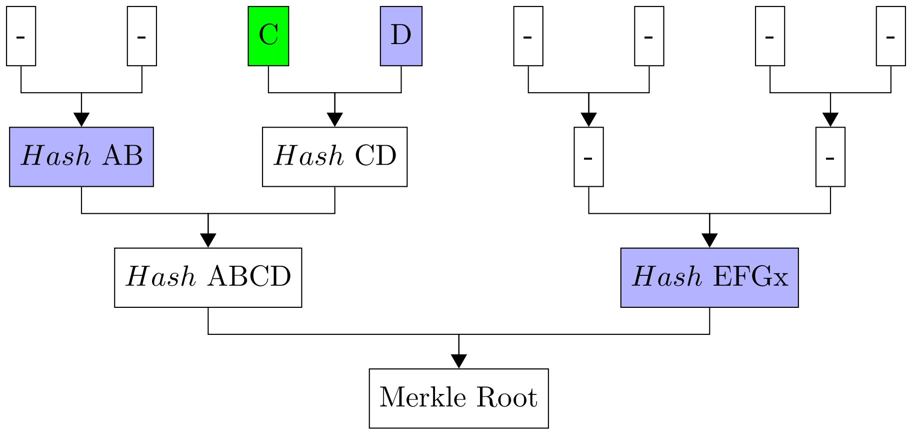

Diagram 6-2: Proof of Membership

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 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.

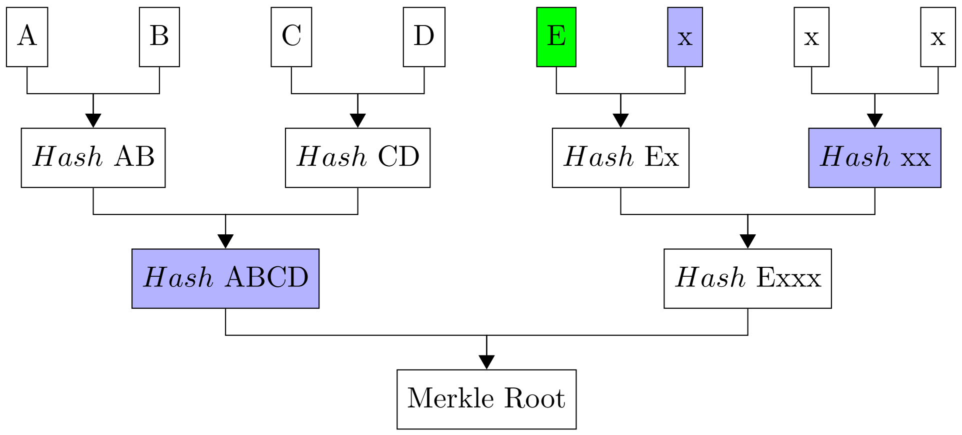

Diagram 6-3: Highest Element Membership Proof

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).

[MC-tx] src/membership_proofs/mod.rs **hash_nil()**

[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 **??**).

*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 **??**). 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.

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.
3. Proof elements: one per node in the proof.[6]
   (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.[7,8]
   (b) `hash`: The node hash h^{node}, used in combination with other node hashes to reconstruct the proof's root hash.

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

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

[6] 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.

[7] The hash function for making nodes is $\mathcal{H}_{Blake2b256}()$ (which was mentioned in footnote 4 of Section 4.2). 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 [38] provided by the `dalek-cryptography merlin` library [37].

[8] 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.

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

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*, 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` [79]. Its name is `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 **??**.

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 **??** for more on fees).

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

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 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:

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

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

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 jth 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()`

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

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 [89]. 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 [89] (used in Monero [79]).

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

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 [93] and Monero [120]. 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 **??** and **??**).

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

[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`, `TxOutMembershipProofs` 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 **??** regarding this terminology.

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

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 $(\sum_j C'^a_j \overset{?}{=} \sum_t C^b_t + fH)$.

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 **??**). 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 $(16 + n * (16 + 32))$ bytes.

Each ring member has a membership proof and 188-byte output copy, so a transaction input is $(v+1) * (204 + 48 * n)$ 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 [103, 119].

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 [87]).

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 **??**.

- 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*: $b_t \oplus_8 \mathcal{H}_n(\text{“}mc\_amount\_value\text{”}, \mathcal{H}_n(r_t K_t^{v,i}))$
  - **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 **??**), so a fee of 1.0 would be recorded as 1000000000000.
- Tombstone block: A future block’s index selected by the transaction author. If the transaction has not been added to the blockchain by the time its tombstone block has been created, then the transaction ‘dies’ and is no longer eligible for adding to the chain.[10,11]

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

[10] In the initial version of MobileCoin, a transaction must be submitted within 100 blocks of its tombstone block, otherwise it will be considered invalid. This requirement will likely inhibit applications of multisignaures [79] 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 **??** for a use-case for tombstone blocks.

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 [20, 31]), while Chapter **??** 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 [31].

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 [20]. 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. [31]

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.

[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 [11] to process the message chunks, and key exchange for creating the AES 'cipher key' uses the Noise Protocol Framework [100] (which involves several Diffie-Hellman exchanges). A communication channel where both parties have the same private key (e.g. shared secret) is called *symmetric encryption* [31].

8.1.3 Remote attestation

The second problem faced by data owners is 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'. [31]

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 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 [40], 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 [67, 69]. Unfortunately enclaves are quite complicated, involve a lot of hardware-related implementation details, and have incomplete documentation [20], 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 [68].

ECALLs/OCALLs for transitioning execution between an enclave and its host process are not addressed). See [31] (especially chapter 5), [112], and [69] 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' ([31] 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 ([31] 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 ([31] 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) [60] 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 ([31] section 6.1.2). Aside from hardware secrets, all other parts of the SGX technology are implemented in microcode ([31] 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] ([72] section 2.2)

[3] Microcode is essentially processor firmware [39], meaning it is permanent or semi-permanent executable code embedded in the processor [5]. Its purpose is translating higher level CPU instructions into hardware-level CPU operations [116].

[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 ([31] section 6.1). Since SGX is so complex, it is prone to vulnerabilities [7] 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. ([31] section 6.6.2 and [32] section 2.6)

[6] Intel documentation [72] labels what [31] 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. ([31] section 6.2)

Aside from the `ELRANGE` memory which stores an enclave's code and data, there is also a separate ([31] 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. ([31] 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 ([31] 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 [[31] 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 [69] 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 ([31] 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 ([72] section 2.1).[9]

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

Table 8-1 summarizes the possible key derivation material ([31] 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 [31] and [69].

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. ([72] 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. ([31] section 5.7.5)

[11] SGX keys can only be created if the `INIT` flag is set and the `DEBUG` flag is not. ([31] 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. ([31] section 5.7.5)

[13] The `CPUSVN` is supposedly a concatenation of security versions corresponding to different components of the SGX implementation. ([31] section 5.7.3)

[14] Although not stated explicitliy, [72] 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. ([31] 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) ([31] 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 ([69] 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 ([31] 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 ([31] section 5.7.5), or long-term storage of secrets ([72] 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 **??** 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 [72] and table 40-64 of [69]) 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 [72] and table 40-64 of [69]) implies `EGETKEY` generates `KEYID` for the Provisioning and Provisioning Seal keys (by stating the value is obtained directly). However, in ([31] 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 [68] 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. ([31] 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). They 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 [69] pg. 40-111 thru 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 ([69] 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 [31] 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) ([69] 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. ([31] 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 [74] RSA signature scheme [[31] section 5.7.1]).[25] Users must pass that `SIGSTRUCT` (laid out in Table 8-4 [[31] 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 [31] 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 ([31] 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 ([69] 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'. ([31] 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 ([31] 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 [[31] 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` 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. ([31] 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 [[31] section 5.2.5]), `ELRANGE` size (number of EPC pages allocated to the enclave), padding}.[29] ([31] 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 ([31] 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. ([31] section 5.3.2)

  Once a new page is prepared, it is copied from non-enclave memory into the appropriate EPC page by `EADD`. ([69] 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] ([31] 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. ([69] 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` [[69] pg. 40-45][31]), padding}, and the remaining four blocks divide up the 256-byte chunk. ([31] 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.

The point of restricting which enclaves can make these keys is they don't include the `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. ([31] section 5.7.5)

Therefore, it is beneficial to owners of SGX-enabled hardware to limit which enclaves may have the `PROVISIONKEY` attribute. As [31] 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`) or author (`MRSIGNER`) are not in a pre-approved list. Unfortunately, Intel does not allow owners

[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 sample code on page 40-45 of [69] 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 [31] sections 5.3.2 and 5.6.4.

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 ([6], [31] 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' [73] and implement a custom Launch enclave [6]. Based on [6] it seems FLC-type Launch enclaves do not require approval from Intel. We may investigate FLC and related techniques [106] 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. ([31] section 5.9.1)

[34] It is not made clear in [31] or [69] if the Launch enclave contains a hard-coded Intel key (or set of keys), or 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

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 ([69] 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 the `SIGSTRUCT`'s `ENCLAVEHASH` matches the final `MRENCLAVE` value.

   (c) Check 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 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:

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

constraints on what the real attributes may be) or the intended attributes. The FLC-type reference Launch enclave implementation at [68] `/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). ([31] section 5.6)

   (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 [27]) 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 3). We recommend [70] section 2.4.5.1, [114], and especially [68] for a more accurate and complete picture of the provisioning process than is laid out here.

[36] According to [70] and [114], provisioning is designed by Intel to use two enclaves (the 'Provisioning enclave' and 'Provisioning Certification enclave'), however [15] claims (without direct citation) these are implemented in practice as one combined enclave. It seems at least Intel's Linux implementation uses separate enclaves [68]. 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 [68].

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 [114] 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' ([27], [70] section 2.4.5.1, [114]). 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 Seal by the hardware (which only the hardware can decrypt), allowing secure key recovery at a later date ([70] section 2.4.5.1).[38]

[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.

[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 ([70] 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 [72].

Attestation keys become ‘members’ of a group public key for the EPID (Enhanced Privacy ID) signature scheme [27] after ‘joining’ the group. The EPID Join protocol [27] 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 ([31] section 5.8.2).[41] EPID permits the issuer (Intel) to reject attestation signatures produced by compromised group members [27, 72].[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 ([31] section 5.8.2).[43]

[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 ([70] 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 [27].

[41] EPID signatures can be linkable or unlinkable (i.e. one signature linkable to another signature), depending on the hardware operator’s preference [72, 20]. MobileCoin nodes and Fog service operators currently use the linkable variant by default [92], which makes it easier to identify and revoke compromised Attestation keys [72]. 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 [27] 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 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.

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 [8] or Noise [100] 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, so with the channel key found in just one `Attestation Verification Report`, any user can connect to an enclave remotely.

SGX local attestation

Attestation between enclaves on the same machine relies on the `EREPORT` SGX CPU instruction ([31] 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).

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 `MAC`s 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]

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

[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`.

Table 8-6 itemizes the content of a `Report` (see [31] section 5.8.1, or [66] 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 [66].

1. The Quoting enclave replaces the `Report`'s `MAC` with an EPID signature using the Attestation key ([31] 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. Intel re-signs the `Quote` with an Intel RSA key, transforming it into an `Attestation Verification Report` (`AVR`) [66].
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.[47] 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()`

[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 `Report`s become worthless after every SGX implementation security update ([31] section 5.8.1).

[47] In reality, `Attestation Verification Reports` 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. [66]

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

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] Extended Euclidean Algorithm. Brilliant.org. https://brilliant.org/wiki/extended-euclidean-algorithm/ [Online; accessed 12/08/2020].

[4] Fermat's Little Theorem. Brilliant.org. https://brilliant.org/wiki/fermats-little-theorem/ [Online; accessed 12/02/2020].

[5] Firmware. https://simple.wikipedia.org/wiki/Firmware [Online; accessed 11/10/2020].

[6] 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].

[7] Software Guard Extensions: Attacks. https://en.wikipedia.org/wiki/Software_Guard_Extensions#Attacks [Online; accessed 11/09/2020].

[8] Transport Layer Security (TLS). IEEE. https://site.ieee.org/ocs-cssig/?page_id=658 [Online; accessed 12/01/2020].

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

[10] XOR – from Wolfram Mathworld. http://mathworld.wolfram.com/XOR.html [Online; accessed 04/21/2018].

[11] 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].

[12] Fungible, July 2014. https://wiki.mises.org/wiki/Fungible [Online; accessed 03/31/2020].

[13] 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].

[14] Base58Check encoding, November 2017. https://en.bitcoin.it/wiki/Base58Check_encoding [Online; accessed 02/20/2020].

[15] 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].

[16] 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.

[17] 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].

[18] Jean-Philippe Aumasson, Willi Meier, Raphael Phan, and Luca Henzen. *The Hash Function BLAKE*. Springer Publishing Company, Incorporated, 2014.

[19] 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].

[20] JP 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].

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

[22] 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].

[23] 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].

[24] 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].

[25] 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].

[26] Dan Boneh and Victor Shoup. A Graduate Course in Applied Cryptography. https://crypto.stanford.edu/~dabo/cryptobook/ [Online; accessed 12/30/2019].

[27] 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].

[28] 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].

[29] 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].

[30] 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].

[31] 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].

[32] 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].

[33] dalek cryptography. Bulletproofs. https://doc-internal.dalek.rs/bulletproofs/index.html [Online; accessed 03/02/2020].

[34] dalek cryptography. bulletproofs [library]. https://github.com/dalek-cryptography/curve25519-dalek [Online; accessed 10/06/2020].

[35] dalek cryptography. bulletproofs [library]. https://github.com/dalek-cryptography/bulletproofs [Online; accessed 10/06/2020].

[36] dalek cryptography. ed25519-dalek [library]. https://github.com/dalek-cryptography/ed25519-dalek [Online; accessed 01/06/2021].

[37] dalek cryptography. merlin [library]. https://github.com/dalek-cryptography/merlin [Online; accessed 10/15/2020].

[38] Henry de Valence. Merlin Transcripts. https://merlin.cool/index.html [Online; accessed 10/15/2020].

[39] debian wiki. Microcode. https://wiki.debian.org/Microcode [Online; accessed 11/10/2020].

[40] debian wiki. ReproducibleBuilds About. https://wiki.debian.org/ReproducibleBuilds/About [Online; accessed 10/29/2020].

[41] Google Developers. Protocol Buffers Encoding. https://developers.google.com/protocol-buffers/docs/encoding [Online; accessed 01/07/2021].

[42] RustCrypto Developers. RustCrypto: BLAKE2 [library]. https://docs.rs/crate/blake2/0.9.0 [Online; accessed 10/06/2020].

[43] W. Diffie and M. Hellman. New Directions in Cryptography. *IEEE Trans. Inf. Theor.*, 22(6):644–654, September 2006. https://ee.stanford.edu/~hellman/publications/24.pdf [Online; accessed 03/04/2020].

[44] 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].

[45] 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].

[46] 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].

[47] 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].

[48] 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].

[49] 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].

[50] 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].

[51] Forcepoint. What is Defense in Depth? https://www.forcepoint.com/cyber-edu/defense-depth [Online; accessed 10/23/2020].

[52] 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].

[53] 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].

[54] Joshua Goldbard and Moxie Marlinspike. MobileCoin, November 2017. https://static.coinpaprika.com/storage/cdn/whitepapers/4235403.pdf?__cf_chl_jschl_tk__=4f3d03da0f296c2787897377ab0da28fdf6abece-1605734577-0-ARyN9UM60LpEfrsQ3ltzLLl__jZTGHp4slF4DQHrW68JD3cCb-4oYKgWr6bHduNKB5A_

qhENHMOLZAXG4k9wWAB0XNIBdmwsdaW8CeOmTRQhE06BsL6FII7hTvaN4SrdrNjFmuTYGp43IqKtkg4LK5duDR59GpglMgTD5UObi3Frxp8Am
bGeCyuw_Ki16QOOlMkFM-oY4pydw3DfaHWNjmN4ts5SHXW38st0XVHCQi_M__wzr9PtEG3BnBzQsmFDmESrQRZIYQXc_
NpDNzpTPihGzETUVL6okA0zTFuIjX6ztkhxnTT4au6BSTzEA2KZpR7s_ts1qlZ6aw1qp7vsqqxBUY9W-si68n0hNd7ro_
wGQuVe7BaNXGwhCOIDfRvYRqKP2vitUCG-_Gs4hkpwC1bt6w [Online; accessed 12/02/2020].

[55] The Ristretto Group. Hash-to-Group with Elligator. https://ristretto.group/formulas/elligator.html [Online; accessed 10/08/2020].

[56] The Ristretto Group. Ristretto. https://ristretto.group/ristretto.html [Online; accessed 10/05/2020].

[57] The Ristretto Group. Ristretto in Detail: Equality Testing. https://ristretto.group/details/equality.html [Online; accessed 10/06/2020].

[58] The Ristretto Group. What is Ristretto? https://ristretto.group/what_is_ristretto.html [Online; accessed 10/06/2020].

[59] The Ristretto Group. Why Ristretto? https://ristretto.group/why_ristretto.html [Online; accessed 10/03/2020].

[60] 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].

[61] 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].

[62] Mike Hamburg. Decaf: Eliminating cofactors through point compression, 2015. https://www.shiftleft.org/papers/decaf/decaf.pdf [Online; accessed 10/05/2020].

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

[64] 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].

[65] imperva. Defense-in-Depth. https://www.imperva.com/learn/application-security/defense-in-depth/ [Online; accessed 10/23/2020].

[66] 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].

[67] Intel. Intel Software Guard Extensions. https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html [Online; accessed 11/07/2020].

[68] Intel. linux-sgx [library]. https://github.com/intel/linux-sgx [Online; accessed 10/29/2020].

[69] 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].

[70] 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].

[71] 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].

[72] 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].

[73] 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].

[74] 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].

[75] 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].

[76] 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].

[77] Bradley Kjell. Big Endian and Little Endian. https://chortle.ccsu.edu/AssemblyTutorial/Chapter-15/ass15_3.html [Online; accessed 01/23/2020].

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

[79] 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].

[80] 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].

[81] Ben Lynn. Lagrange's Theorem. https://crypto.stanford.edu/pbc/notes/group/lagrange.html [Online; accessed 12/02/2020].

[82] 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].

[83] Gregory Maxwell. Confidential Transactions. https://elementsproject.org/features/confidential-transactions/investigation [Online; accessed 11/23/2020].

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

[85] 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].

[86] 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].

[87] 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].

[88] 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].

[89] 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.

[90] 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].

[91] 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].

[92] MobileCoin. MobileCoin Consensus Service. https://github.com/mobilecoinfoundation/mobilecoin/blob/master/consensus/service/README.md [Online; accessed 02/01/2021].

[93] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008. http://bitcoin.org/bitcoin.pdf [Online; accessed 03/04/2020].

[94] 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].

[95] Sarang Noether. Janus mitigation, Issue #62, January 2020. https://github.com/monero-project/research-lab/issues/62 [Online; accessed 02/17/2020].

[96] 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].

[97] 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].

[98] 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].

[99] 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].

[100] Trevor Perrin. The Noise Protocol Framework, July 2018. https://noiseprotocol.org/noise.pdf [Online; accessed 11/06/2020].

[101] Shaan Ray. Merkle Trees, December 2017. https://hackernoon.com/merkle-trees-181cb4bc30b4 [Online; accessed 10/20/2020].

[102] 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].

[103] Neptune Research and Isthmus. Monero tx_extra statistics, September 2020. https://github.com/neptuneresearch/monero-tx-extra-statistics-report [Online; accessed 10/13/2020].

[104] 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].

[105] SafeCurves. SafeCurves: choosing safe curves for elliptic-curve cryptography, 2013. https://safecurves.cr.yp.to/rigid.html [Online; accessed 03/25/2020].

[106] 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].

[107] C. P. Schnorr. Efficient Identification and Signatures for Smart Cards. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 239–252, New York, NY, 1990. Springer New York. https://link.springer.com/content/pdf/10.1007%2F0-387-34805-0_22.pdf [Online; accessed 03/04/2020].

[108] 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].

[109] Bassam El Khoury Seguias. Monero Building Blocks, 2018. https://delfr.com/category/monero/ [Online; accessed 10/28/2018].

[110] 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].

[111] Victor Shoup. *A Computational Introduction to Number Theory and Algebra*, pages 15–16. Cambridge University Press, 2005.

[112] SSLab. SGX 101. https://sgx101.gitbook.io/sgx101/ [Online; accessed 11/07/2020].

[113] Andrew V. Sutherland. Schoof's algorithm, March 2015. https://math.mit.edu/classes/18.783/2015/LectureNotes9.pdf [Online; accessed 12/02/2020].

[114] 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].

[115] 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].

[116] 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].

[117] 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].

[118] Certificate Transparency. How Log Proofs Work. http://www.certificate-transparency.org/log-proofs-work [Online; accessed 10/20/2020].

[119] UkoeHB. Proposal/Request: Update Supplementary Transaction Content, Issue #6456, April 2020. https://github.com/monero-project/monero/issues/6456 [Online; accessed 10/11/2020].

[120] Nicolas van Saberhagen. CryptoNote V2.0. https://cryptonote.org/whitepaper.pdf [Online; accessed 04/04/2018].

[121] Adam "waxwing" Gibson. From Zero (Knowledge) To Bulletproofs, June 2020. https://github.com/AdamISZ/from0k2bp/blob/master/from0k2bp.pdf [Online; accessed 01/13/2021].

[122] 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].

[123] Wikibooks. Undergraduate Mathematics/Coset, 2020. https://en.wikibooks.org/wiki/Undergraduate_Mathematics/Coset [Online; accessed 10/06/2020].