MONERO

PROTOCOL GUIDE

Monero in depth

Complete guide

This guide describes the Monero mainnet protocol (November 2026) in enough detail to understand each of its pieces: primitives, transactions, consensus and network.

From the basics to the cryptography

Contents

INTRODUCTION AND BASICS

· How to use this guide 3

· The basic concepts 3

· Cryptography from scratch 5

CHAPTERS

01 Overview and threat model 9

· Notation 11

02 Cryptographic foundations 11

03 Keys, addresses and subaddresses 14

04 Stealth outputs (stealth addresses) 16

05 Key images and double-spend prevention 19

06 Ring signatures: from MLSAG to CLSAG 21

07 Decoy selection 24

08 RingCT: confidential amounts 26

09 Range proofs: Bulletproofs+ 28

10 Anatomy of a transaction 30

11 Consensus and mining 33

12 P2P network and transport layer 35

13 Wallets in practice 37

14 Attacks and the limits of privacy 38

15 The future: FCMP++ and Carrot 40

16 Lab: seeing the protocol at work 42

17 References and recommended reading 45

INTRODUCTION

How to use this guide

Audience and prerequisites. You do not need to know any cryptography or blockchain: the "Before you start" sections explain the basic concepts and the minimal maths from scratch. After that the guide climbs in level until it reaches the formulas.

State of the protocol. Today the mainnet uses CLSAG signatures with rings of 16 members and Bulletproofs+ range proofs. FCMP++ and Carrot are being tested and have no activation date; they are covered in chapter 15.

How to read it. If you are starting from zero, begin with "Before you start" and consult the notation table, just before chapter 2, when you reach the formulas. Each chapter opens with a sentence that sums up the idea; the key formulas are marked, and each comes with its explanation in words, the meaning of every symbol and, when it helps, an example with small numbers. Chapters 2 to 10 are the cryptographic core and should be read in order. Chapters 11 to 14 (consensus, network, wallets, attacks) can be read separately. Chapter 16 offers code exercises.

BEFORE YOU START

The basic concepts

To understand Monero you need five prior ideas: what a blockchain is, what keys and signatures are, what a hash is, why Bitcoin is not private, and what a Monero payment looks like from a bird's-eye view. If you already know them, jump to chapter 1.

From a central ledger to a shared one

At a bank, a single entity keeps the ledger that records who owns how much, and you trust it. A cryptocurrency replaces that ledger with a **shared** one: thousands of computers (the *nodes*) hold an identical copy and agree on its contents without anyone being in charge.

- Operations are grouped into **blocks**. Each block includes the fingerprint (hash) of the previous one, so the blocks form a **chain**: the *blockchain*.

- Changing an old operation would force you to redo that block and all the following ones, and the network would notice.
- **Miners** compete to add the next block by solving a problem that takes a lot of computation (*proof of work*). The winner collects new coins plus the fees. Rewriting history would take more power than all the other miners put together.

Keys and signatures

Every user has two linked numbers. The **private key** is secret and is used to authorize payments. The **public key** is derived from it and can be shared; it works like a mailbox number where anyone can drop money for you.

A **digital signature** proves that whoever made it knows the private key, without revealing it, and it is tied to one specific message (the transaction). If someone changes a single digit of the transaction, the signature stops being valid. The key property is that getting the public key from the private key is easy, but the reverse path is infeasible.

Hashes

A **hash** turns any data into a fixed-size fingerprint. It always gives the same result for the same input, changes completely if you modify a single bit, and cannot be reversed. It is used to link blocks, identify transactions, derive keys and build the mining problem.

Coins as "outputs", not balances

Bitcoin and Monero have no accounts with a balance. They have **outputs**: each one is an amount locked to a specific key, like loose banknotes. To pay, your wallet spends one or more whole outputs and creates new ones: one for the recipient and one with the change for you.

A spent output cannot be spent again. Preventing that **double spend** is the central problem of any digital currency, and Monero solves it in a special way (ch. 5).

Why Bitcoin is not private

In Bitcoin everything is public: which address pays which, how much, and which outputs are spent. There are no names, but a single purchase made with your identity at one point is enough for chain-analysis companies to follow the money forwards and backwards. Being pseudonymous is not the same as being private.

That has a second consequence, **fungibility**: if a coin's history is visible, a coin with a "bad reputation" can be worth less than an identical one. Money should be interchangeable unit for unit.

A Monero payment from a bird's-eye view

1. The recipient gives you their **address**, which is long and reusable.
2. Your wallet uses that address to create a **new, unique destination** for this payment, which only the recipient can recognize as theirs. Their address does not appear on the chain (*stealth address*, ch. 4).
3. To spend your coins, the wallet puts them in a group with 15 other outputs from the chain that belong to other people and signs in a way that proves *one* of the group is yours, without saying which (*ring signature*, ch. 6). A unique tag per output (*key image*, ch. 5) prevents it from being spent twice.
4. The amounts go in "sealed envelopes" (*commitments*) that the network can add up without opening, to check that no money is created. An extra proof ensures that no envelope contains a negative value (chs. 8 and 9).
5. Miners include the transaction in a block and every node verifies it.

The result is privacy at three levels at once: nobody knows who receives, who sends or how much, and everything remains verifiable by anyone.

BEFORE YOU START

Cryptography from scratch

All of Monero's privacy comes from a few mathematical ideas, and all of them can be understood with clocks, paint and envelopes. No prior knowledge is assumed here.

Clock arithmetic

On a 12-hour clock, 9 + 5 is not 14 but 2: once you pass 12 you start over. That is **modular arithmetic**. Cryptography uses the same thing with giant clocks, with numbers of about 77 digits, so that every operation gives a result that looks random.

The key is that some operations are easy in one direction and practically impossible in the other. Example with a 23-hour clock: computing `5^6` and keeping the remainder gives 8 right away. Finding out *which exponent* gives 8 can only be done by trying them one by one, and with 77-digit clocks trying them all would take longer than the age of the universe.

Elliptic curves without fear

An **elliptic curve** is a set of points with a rule for "adding" two points to get a third. You do not need to know geometry: it is enough to accept the rule, just as you accept that on a clock 9 + 5 = 2.

A fixed, public point is chosen, the **base point** `G`. Adding `G` to itself `k` times gives another point, written `k·G`. There are two fundamental facts:

- **Computing `k·G` is fast**, even if `k` has 77 digits: about 250 steps are enough.
- **Recovering `k` from `k·G` is infeasible.** This is the *discrete logarithm problem*, and no shortcut is known for the curve Monero uses.

The whole key scheme comes from this: your **private key** is a number `k` and your **public key** is the point `K = k·G`. You can publish `K` safely.

Convention of this guide: lowercase letters (`a`, `k`, `x`) are numbers; uppercase letters (`A`, `K`, `P`) are points. You can add points and multiply a number by a point, but you cannot multiply two points. Monero uses a specific curve called **Ed25519**.

One property will be very useful later, **linearity**: `(a + b)·G = a·G + b·G`. It lets you combine secret values by operating only with their public versions.

Sharing a secret in public (Diffie-Hellman)

The classic analogy is paint. Alice and Bob agree on a base colour, each mixes in a secret colour of their own, and they swap the result. Then each adds their own secret to the other's mixture and both end up with the same final colour. Someone who watched the exchange cannot "unmix" anything.

With curves it is the same. Alice has `a` and publishes `A = a·G`. Bob has `r` and publishes `R = r·G`. Then:

KEY FORMULA.

$$r \cdot A = r\,a\,G = a\,r\,G = a \cdot R$$

In words: Bob multiplies Alice's public key by his secret number, Alice multiplies Bob's by hers, and the two reach the same point without having revealed anything.

- `a` and `r`: Alice's and Bob's secret numbers. They never travel over the network.
- `G`: the base point of the curve. Everyone knows it and it is the common starting point.
- `A = a·G` and `R = r·G`: the public keys. They can be published because, from them, recovering `a` or `r` is infeasible.
- `r·A` and `a·R`: what each one computes with their secret and the other's public key.

Why it works. The order of the multiplications does not matter: `r·(a·G)` and `a·(r·G)` are both `(a·r)·G`. Someone who only sees `A` and `R` would have to undo one of the multiplications, and that is the discrete logarithm problem, for which no efficient solution is known.

Example with small numbers. With a 17-hour clock and `g = 3` (here "multiplying" is written as a power): Alice chooses `a = 4` and publishes `3^4 mod 17 = 13`. Bob chooses `r = 6` and publishes `3^6 mod 17 = 15`. Alice computes `15^4 mod 17 = 16` and Bob computes `13^6 mod 17 = 16`. Same number, and someone who only saw 13 and 15 did not need to know it. With 17 possibilities you can try them by hand; with the roughly 76-digit numbers of the real curve, you cannot.

Bob computes `r·A` with what Alice published, Alice computes `a·R` with what Bob published, and both obtain the same point. That is their **shared secret**, and nobody else can compute it. Monero uses it so that the sender and the receiver of a payment agree on a key that only they know (ch. 4).

Signatures: proving you know `k` without saying it

A signature must convince anyone that you know `k` such that `K = k·G`, without revealing `k`. The trick, in three steps:

1. **Commitment:** you choose a random number `α` and publish `α·G`.
2. **Challenge:** `c` is computed, a number obtained by hashing everything relevant (including the message). Nobody can choose it in their favour.
3. **Response:** you publish `s = α - c·k`.

The verifier checks an equality that only holds if `s` was computed with the correct `k`:

KEY FORMULA.

$$s\,G + c\,K = (\alpha - ck)\,G + c\,k\,G = \alpha\,G$$

In words: the verifier does not need the secret. With the public key and the numbers in the signature they check that everything adds up, and only someone who knew the private key could have achieved that.

- `k` and `K = k·G`: the private key and its public key.
- `α`: the random number that covers the secret; without it, `s` would give `k` away.
- `c`: the challenge, which comes from a hash of the message and of `α·G`; nobody can choose it in their favour.
- `s = α - c·k`: the response that gets published.

Example with small numbers (treating `G` as a unit): with `k = 3` and `α = 7`, and a challenge `c = 2`, the response is `s = 7 - 2·3 = 1`. The verifier, who only knows `K = 3G`, computes `1·G + 2·K = 1G + 6G = 7G`, which is `α·G`, exactly what was published at the start. If someone did not know `k`, they could not fabricate an `s` that fits.

Since `s` is masked by the random number `α`, it does not reveal `k`. And since `c` depends on the message, the signature is only valid for that transaction. Monero's ring signatures are a more elaborate version of this same scheme.

Commitments: the sealed envelope

A **commitment** is a sealed envelope: it ties you to a value without revealing it, and later you can open it without anyone doubting what it contains. A good commitment does two things: it **hides** the value and it is **binding** (you cannot change it afterwards).

Monero uses the Pedersen commitment: `C = y·G + a·H`, where `a` is the amount, `y` is a random number that acts as the opaque paper of the envelope, and `H` is a second fixed point. Thanks to linearity, **adding two envelopes gives the envelope of the sum**, and that makes it possible to check that inputs equal outputs without opening any of them (ch. 8).

Zero-knowledge proofs

A **zero-knowledge** proof convinces you that something is true without showing why. You have already seen one: the signature proves that you know `k` without showing it. Monero uses two others: "my output is one of these 16" (ring signature) and "this amount is between 0 and 2^64" (range proof).

Quick glossary

TERM	IN ONE SENTENCE
Node	Computer that stores the chain and verifies all the rules
Miner	Whoever proposes blocks by solving the proof of work
Wallet	Program that stores your keys, creates transactions and scans the chain
Output	Amount of coins locked to a key; what gets spent
Private / public key	Secret number `k` and its point `k·G`, which can be shared
Spend key / view key	Two Monero keys: one to spend, the other only to see what has been received
Stealth address	One-time destination derived from your address for each payment
Ring signature	Signature proving that one of several keys signed, without saying which
Key image	Unique tag per spent output that prevents double spending
Pedersen commitment	Sealed envelope for an amount, which can be added without opening
Range proof	Proof that a hidden amount is neither negative nor gigantic
Mempool	Waiting room for transactions not yet included in a block
Hard fork	Protocol upgrade that all nodes must adopt from a certain block
Piconero	Smallest unit: 1 XMR = 10^12 piconero

CHAPTER 1

Overview and threat model

Monero hides three things by default in every transaction (receiver, sender and amount) and leaves everything else public. Understanding what remains visible is as important as

understanding what is hidden.

WHAT IS HIDDEN	MECHANISM	WHAT STAYS VISIBLE ON THE CHAIN
Receiver	Stealth addresses (ch. 4)	A one-time key `P` per output and the transaction public key `R`
Sender	CLSAG ring signature (ch. 6)	The 16 ring members and the key image (ch. 5)
Amount	Pedersen commitments and Bulletproofs+ (chs. 8-9)	The fee, the number of inputs and outputs, the size

Fungibility as a design goal. If every coin has a traceable history, some are worth more than others. Monero aims for outputs to be indistinguishable from one another, which is why privacy is mandatory and not optional.

Adversaries worth telling apart

1. **Passive chain observer.** Has the whole public history and applies statistical analysis to the rings. This is the adversary CLSAG and decoy selection were designed against.
2. **Node operator or network eavesdropper.** Sees which IP a transaction comes from and, if you are a client of a remote node, which outputs you query. The chain does not protect you from this.
3. **Counterparty or exchange.** Knows the address and the moment of a specific payment. It can cross that data with its own view of the chain.
4. **Attacker with their own outputs (EAE).** If they control a large fraction of the outputs used as decoys, they can discard rings and reduce ambiguity (ch. 14).

Costs of privacy

Each transaction weighs around 2 kB, compared with about 200 to 250 bytes for a typical Bitcoin transaction. Nodes cannot know which outputs are spent, so there is no UTXO set that can be pruned. And every wallet has to scan the whole chain with its view key to find its funds.

REFERENCE

Notation

This table fixes the symbols of the formulas that appear from chapter 2 onwards.

SYMBOL	MEANING
`G`	Ed25519 base point
`H`	Second generator, with no known discrete logarithm with respect to `G` (derived by hashing `G`)
`l`	Order of the prime subgroup: 2^252 + 27742317777372353535851937790883648493
lowercase (`a`, `x`)	Scalar modulo `l` (private key or random number)
UPPERCASE (`A`, `P`)	Curve point
`Hs(·)`	Keccak-256 reduced modulo `l` (hash to scalar)
`Hp(·)`	Hash to point, multiplied by 8 to land in the prime subgroup
`‖`	Byte concatenation

Monero uses the original Keccak (pre-NIST padding), not SHA3-256.

CHAPTER 2

Cryptographic foundations

This chapter reviews rigorously what was seen in "cryptography from scratch": Monero's privacy rests on the Ed25519 curve, the Keccak hash, the Diffie-Hellman exchange and Pedersen commitments. Here are the details that matter when implementing them.

Ed25519 and the cofactor

The curve has order `8l`, with `l` prime. The cofactor 8 is a source of subtle bugs, and Monero handles it in three ways:

- Private keys are scalars reduced modulo `l`.
- Diffie-Hellman derivations are multiplied by 8 to eliminate any torsion component.
- Nodes require every key image to lie in the prime subgroup, checking that `l·I` is the identity element. Without that check, a key image could be modified with a torsion point and spend the same output twice.

Hashes

`Hs(m)` is Keccak-256 of `m` reduced modulo `l`. `Hp(P)` applies Keccak-256 to the encoding of `P`, maps it to a curve point with a CryptoNote-specific hash-to-point function, and multiplies it by 8. The result is a point whose discrete logarithm with respect to `G` nobody knows, and that is what makes the key image secure (ch. 5).

Diffie-Hellman

If `A = aG` and `R = rG`, both sides reach the same secret without exchanging private keys:

KEY FORMULA.

$$r \cdot A = raG = arG = a \cdot R$$

In words: it is the same exchange as before with different letters. The sender (`r`) and the receiver (`a`) each combine their secret with the other's public key and reach the same point, which Monero uses as the shared secret of each payment.

The sender computes it with `r` and the receiver's public view key; the receiver, with their private view key and the `R` published in the transaction. It is the core of stealth outputs.

Pedersen commitments

A commitment to the value `a` with blinding factor `y` is:

KEY FORMULA.

$$C = yG + aH$$

In words: the commitment is a sealed envelope that holds the amount together with a secret random number: everyone sees the envelope, nobody sees the amount, and the author cannot change it afterwards.

- `a`: the amount you want to hide.
- `y`: the blinding factor, a secret random number that acts as the opaque paper of the envelope.
- `G`: the public base point.
- `H`: a second public point, chosen so that nobody knows how it relates to `G`; that is what prevents changing the amount later.
- `C`: the commitment, a curve point that is published on the chain.

Why it works. It hides because `y·G` is a uniform point that covers the amount. It is binding because opening it to another amount would require knowing the relationship between `G` and `H`. And it is linear: adding two commitments gives the commitment of the sum of the amounts.

Toy example (numbers on a 101-hour clock, just to see the addition; not secure): with `G = 2` and `H = 7`, the amount 5 with `y = 10` gives `C = 10·2 + 5·7 = 55`, and the amount 3 with `y = 4` gives `C = 4·2 + 3·7 = 29`. Added together they give `84`, which is exactly the commitment to the amount `8` with factor `14`: `14·2 + 8·7 = 84`. That is how you check that the books balance without opening any envelope.

- **Hiding:** without `y`, `C` does not reveal `a`, because `yG` is a uniform point.
- **Binding:** nobody can open `C` to two different values without knowing `log_G(H)`.
- **Homomorphic:** `C(y1, a1) + C(y2, a2) = C(y1 + y2, a1 + a2)`. Nodes verify sums of amounts without seeing them.

Schnorr-type signatures and proofs of knowledge

All of Monero's signatures (ring, range) are variants of the proof of knowledge of a discrete logarithm: commitment, a challenge derived by hashing (Fiat-Shamir) and a response. If you remember the structure `s = α - c·x`, you already have half of CLSAG.

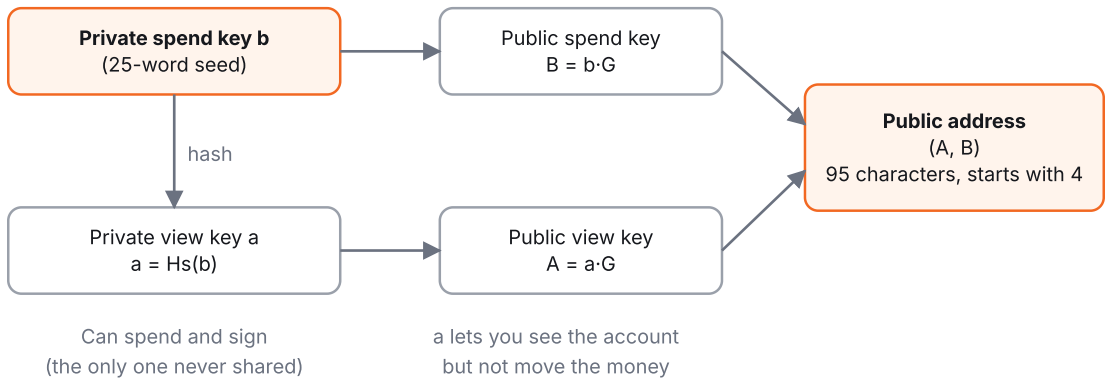
CHAPTER 3

Keys, addresses and subaddresses

A Monero account is two key pairs, and everything else (seed, address, subaddresses) is derived from them deterministically.

Key hierarchy

A Monero account rests on two private keys: one is used to spend and the other to view. The view key is obtained from the spend key, so a single backup recovers everything. Someone who only has the view key can look at the account but cannot move the money.



The view key and the public address both come from the spend key.

$$b \in \mathbb{Z}_l \text{ (private spend key)} \qquad B = bG$$
$$a = H_s(b) \text{ (private view key)} \qquad A = aG$$

In words: the view key comes from passing the spend key through a hash function, and each private key has its public key, which is the one that gets shared (`B` for spending, `A` for viewing).

- **Spend (`b`):** lets you sign and spend. It is the only key that must always be protected.
- **View (`a`):** lets you detect incoming funds and decrypt amounts, but not spend. It is used for read-only wallets and audits.

- The classic mnemonic seed is 25 words: 24 encode the 256 bits of `b` and the last one is a checksum. Newer formats exist, such as Polyseed (16 words with a creation date), which some wallets already support.

Standard address

The address is the concatenation encoded in base58 (in blocks, not Bitcoin's base58):

```
network byte (0x12 mainnet) ‖ B ‖ A ‖ checksum (first 4 bytes of the Keccak-256 of the above)
```

It is 95 characters long and starts with `4`. Note that the public spend key comes first and the public view key second.

Subaddresses

A subaddress lets you receive at different addresses that cannot be linked to one another on the chain, without scanning with extra keys: the view key `a` is still enough. For account index `i` and subaddress index `j`:

$$
\begin{aligned}
m_{i,j} &= H_s(\text{"SubAddr\0"} \,\|\, a \,\|\, i \,\|\, j) \\
D_{i,j} &= B + m_{i,j}\, G \quad \text{(public spend key)} \\
C_{i,j} &= a \cdot D_{i,j} \quad \text{(public view key)}
\end{aligned}
$$

In words: each subaddress is obtained by adding to your spend key an offset computed with your view key. You can create as many as you like and, from the outside, nobody can tell they belong to the same person.

The subaddress is the pair `(C, D)`, starts with `8` (network byte 0x2a), and the sender treats it as if it were `(A, B)`, with one key difference: they publish `R = r·D` instead of `r·G`. The receiver recovers the output's spend key, `P - Hs(a·R ‖ k)·G`, and looks it up in their table of `D_{i,j}`. By default the wallet precomputes 50 accounts with 200 subaddresses each.

When a transaction pays several recipients and one of them is a subaddress, `R` no longer works for everyone and the transaction includes an additional public key per output (`additional_tx_pub_keys`).

Payment IDs and integrated addresses

A payment ID was an identifier that the sender attached so that a service could tell apart payments to the same address. The 32-byte ones went in the clear and were removed. The 8-byte ones are encrypted with the shared secret and live in integrated addresses (network byte

0x13). Today they are discouraged in favour of subaddresses, because a payment ID is an extra identifier that degrades privacy.

Note for the future: Carrot (ch. 15) changes the key hierarchy and the address format, so what is in this chapter describes the scheme in force on mainnet.

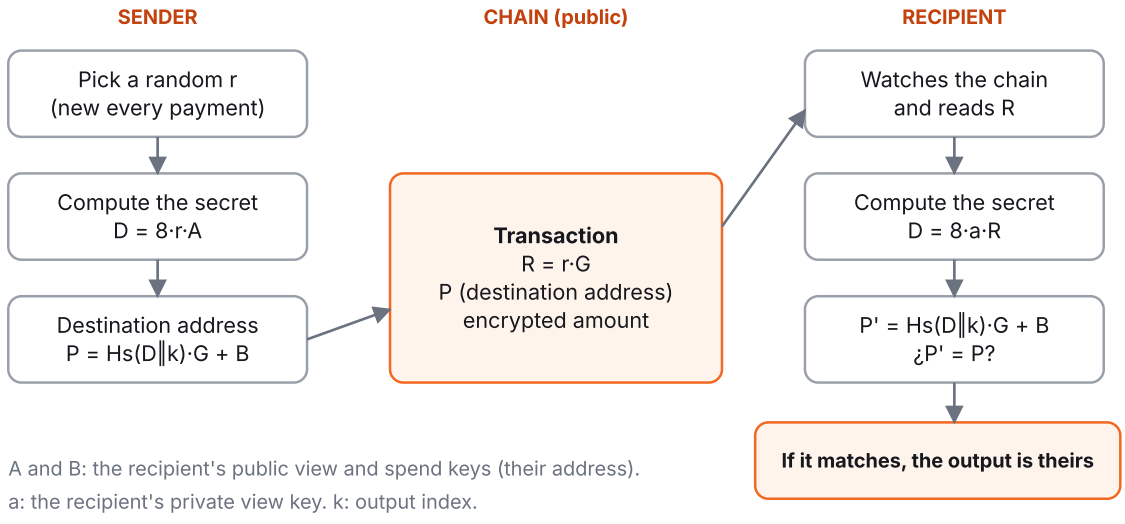
CHAPTER 4

Stealth outputs (stealth addresses)

Each output of a transaction goes to a one-time public key that only the sender and the receiver can relate to the receiver's address. For an outside observer, two payments to the same address are unrelated keys.

Derivation (sender side)

For each payment the sender generates a fresh random number. They combine it with the receiver's public view key to produce a secret that only the two of them can compute, and with that secret they create a one-time destination address for each output. In the transaction they publish only an auxiliary point, which lets the receiver repeat the computation.



How a one-time destination address is created and recognized.

For the output with index `k`, with a random `r` per transaction and the receiver's address `(A, B)`:

KEY FORMULA.

$$R = rG \quad \text{(transaction public key, goes in } \texttt{tx_extra}\text{)}$$
$$D = 8\,rA \quad \text{(derivation; the 8 removes the torsion)}$$
$$P_k = H_s(D \,\|\, k)\,G + B$$

In words: the sender publishes an auxiliary point, combines it with the receiver's view key to obtain a shared secret, and uses it to derive the destination address of each output.

- `r`: a new random number in every transaction; it is the sender's secret.
- `R = r·G`: auxiliary point published in the transaction (the `tx_extra` field).
- `A`, `B`: the receiver's public view and spend keys, which make up their address.
- `D = 8·r·A`: the shared secret (the Diffie-Hellman from before). The `8` discards a minor weakness of the curve.
- `k`: the output index, so that two outputs of the same payment have different addresses.
- `Hs(…)`: a hash converted into a number.
- `P_k`: the one-time destination address of output `k`: the receiver's key `B` shifted by something only the sender and receiver know.

Why it works. An observer sees `R` and `P_k`, but without `a` or `r` they cannot compute `D`, so they cannot relate `P_k` to `B`. Each output looks like a random key with no link to the receiver's address.

Recovery (receiver side)

The receiver repeats the computation from their side. With their private view key and the published auxiliary point they obtain the same secret as the sender, rebuild the destination address and check whether it matches that of the output. If it matches, the output is theirs.

The receiver uses their view key and the transaction's `R`:

KEY FORMULA.

$$
\begin{aligned}
D &= 8\,aR = 8\,arG = 8\,rA \\
P'_k &= H_s(D \,\|\, k)\,G + B \qquad \text{if } P'_k = P_k \Rightarrow \text{the output is mine} \\
x_k &= H_s(D \,\|\, k) + b \qquad \text{(private key to spend it)}
\end{aligned}
$$

In words: the receiver reaches the same secret from the other side, rebuilds the destination address, compares it with the output's and, if it matches, computes the key that lets them spend it.

- `a·R`: the same as `r·A`, because `a·r·G = r·a·G`. That is why sender and receiver obtain the same `D`.
- `P_k'`: the destination address the receiver computes on their own. If it equals the output's `P_k`, the output is theirs.
- `x_k`: the output's private key. It adds the secret offset to `b`, the spend key, which only the owner has: that is why someone who only has the view key detects the payment but cannot spend it.

Only someone with `b` knows `x_k`, so whoever has only the view key can detect the output and read the amount, but not spend it.

What is encrypted with the same secret

From the same shared secret `H_s(D ‖ k)`, using hashes with different tags, both the commitment's blinding factor `y` and an 8-byte mask that is XORed with the amount are derived. The receiver can thus rebuild the commitment and check that it matches the published one (ch. 8).

Scanning cost and view tags

Scanning a transaction costs one scalar multiplication `a·R`, and then one hash and one point addition per output. Over a chain of tens of millions of outputs, this was what took longest when syncing a wallet.

View tags (since the August 2022 hard fork) add to each output a byte `Keccak("view_tag" ‖ D ‖ k)[0]`. The wallet compares that byte first and immediately discards 255 out of every 256 outputs belonging to others without computing `P_k'`. Syncing improves noticeably (on the order of 40%). The cost is that the byte leaks 8 bits about the shared secret, which is considered acceptable because it does not allow a third party to recognize outputs.

One `R` key per transaction

Privacy depends on `r` being new in every transaction. If it were reused, all of that sender's outputs towards the same address would have the same `D` and could be grouped.

CHAPTER 5

Key images and double-spend prevention

The key image is a deterministic tag of each spent output: it does not reveal which one it is, but if it is spent twice the tag appears repeated and the network detects it.

Construction

The key image is a tag that the owner of a coin computes with their secret key. The same coin always produces the same tag, and the tag does not say which coin it comes from. That way, spending the same output twice repeats the tag and the network detects it.

If the output is `P = xG` (with `x` the output's private key, ch. 4), its key image is:

KEY FORMULA.

$$I = x \cdot H_p(P)$$

In words: the key image is the output's private key applied to a point that depends only on that output. It always comes out the same for the same coin, and it cannot be computed without the secret.

- `x`: the output's private key (ch. 4). It is what only the owner knows.
- `P`: the output's public key, with `P = x·G`.
- `Hp(P)`: a hash that turns `P` into another curve point whose relationship to `G` nobody knows. A point other than `G` is used so that `I` cannot be used to deduce `P`.
- `I`: the key image that gets published when spending.

Three properties make it work:

1. **Deterministic.** For a given `x` there is only one `I`. Any attempt to spend `P` again produces the same `I`.
2. **Impossible to compute without `x`.** Someone who sees `P` cannot obtain `I`, because it would require the discrete logarithm of `P`. Someone who sees `I` cannot get to `P` either.
3. **Verifiable in zero knowledge.** The ring signature (ch. 6) proves that there is a ring member with `P_π = xG` and that *the same* `x` produces `I = x·Hp(P_π)`. It is a proof of equality of discrete logarithms in two bases, `G` and `Hp(P_π)`.

Without the third property the signer could invent a different `I` each time and spend the output multiple times.

What a node verifies

On receiving a transaction, for each input the node checks:

- that `I` is in the prime subgroup (`l·I` is the identity),
- that `I` does not appear in the set of key images already spent on the chain or in another transaction in the mempool,
- that the ring signature is valid for that `I` (ch. 6).

The node keeps the complete list of spent key images in its database (LMDB). That list only grows.

Design consequences

- **There is no useful UTXO set.** Since the node does not know which specific output corresponds to a key image, it cannot mark it as spent or prune it. Every output that has ever existed remains a decoy candidate.
- **The key image is public and permanent.** Whoever has your spend and view keys can rebuild your key images and therefore see what you spent. That is why the view key alone is not enough to see outgoing spends, and why read-only wallets need to import the key images to show the correct balance.
- **In the inputs, the ring's outputs are encoded as relative global indices** (`key_offsets`), which save bytes by turning the list into differences between sorted indices.

CHAPTER 6

Ring signatures: from MLSAG to CLSAG

CLSAG proves, with a single loop of challenges, that the signer knows the key of *one* output in the ring and that the difference between their commitment and the pseudo-commitment is a commitment to zero, without revealing which one it is.

Evolution of the scheme

SCHEME	WHAT IT ADDS	SIGNATURE SIZE PER INPUT (RING OF `N`)
CryptoNote ring signature (2013)	Sender anonymity; key image to prevent double spending	Linear in `n`, several scalars per member
LSAG	Linkable ring signature with a compact proof	`n + 1` scalars
MLSAG (RingCT, 2017)	Multilayer signature: proves key and commitment at once	`2n + 1` scalars (`32(2n+1)` bytes)
CLSAG (October 2020 hard fork)	Merges the two layers into one with aggregation coefficients	`n + 2` elements (`32(n+2)` bytes)

With `n = 16`, an input goes from 1056 bytes with MLSAG to 576 bytes with CLSAG, and it is signed and verified faster.

Starting data

The signer knows which of the ring's keys is theirs. They publish two tags computed with their secrets: the key image, which prevents spending twice, and an auxiliary image used to prove that the amounts add up.

The ring has keys `P_0 … P_{n-1}` and commitments `C_0 … C_{n-1}`. The signer knows the real index `π`, the key `x` with `P_π = xG`, and the scalar `z = y_π - y'` such that `C_π - C' = zG`, where `C'` is the input's pseudo-commitment (ch. 8). They publish the key image and an auxiliary image:

$$I = x \cdot H_p(P_\pi) \qquad D = z \cdot H_p(P_\pi)$$

In words: `I` is the usual key image. `D` is the same idea applied to the secret `z` that links the spent amount to its pseudo-commitment; with it the signature checks inputs and amounts in a single pass.

Aggregating the two layers

CLSAG has to check two things at once: that you know a key in the ring and that the amounts add up. Instead of doing it separately, it mixes them into a single check with two coefficients that come from a hash of all the data.

Two coefficients derived by hashing the whole context (ring, commitments, `I`, `D`, `C'`) combine the key layer and the commitment layer into one:

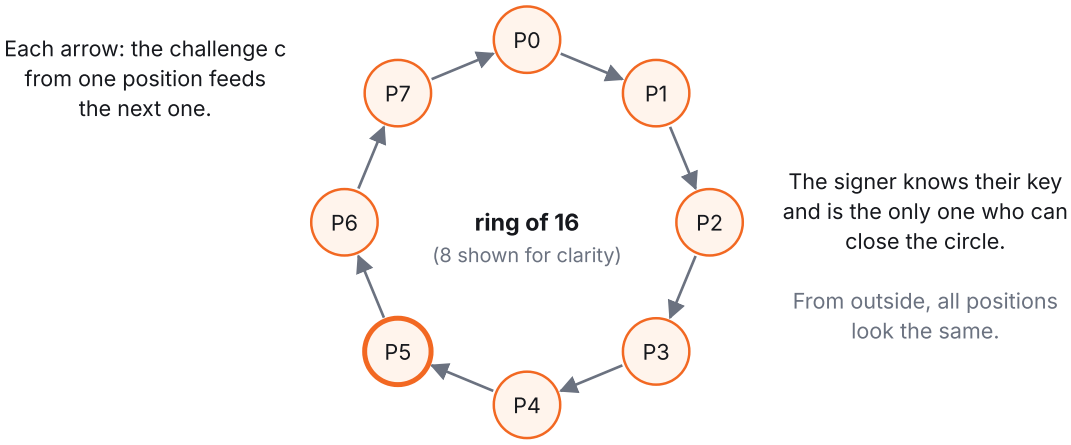
$$\begin{aligned}
\mu_P &= H_s(\text{"CLSAG_agg_0"} \,\|\, P_0..P_{n-1} \,\|\, C_0..C_{n-1} \,\|\, I \,\|\, D \,\|\, C') \\
\mu_C &= H_s(\text{"CLSAG_agg_1"} \,\|\, \text{same as above}) \\
W_i &= \mu_P \, P_i + \mu_C \, (C_i - C') \\
W_I &= \mu_P \, I + \mu_C \, D
\end{aligned}$$

In words: the two `μ` coefficients mix the key part and the commitment part of each ring member into a single combined key `W`. They cannot be chosen at will because they depend on the whole context.

At the real position, `W_π = (μ_P x + μ_C z)·G`: a single secret key `w = μ_P x + μ_C z` controls the whole row.

The challenge loop

The signature is a chain of challenges that closes on itself, like a circle of dominoes: each position in the ring produces a challenge that feeds the next, and only someone who knows the secret key can close the circle. To anyone looking, all positions are indistinguishable.



The ring signature as a closed chain of challenges.

For each index `i` and with the message `m` (hash of the transaction prefix and the rest of the signed data):

KEY FORMULA.

$$L_i = s_i G + c_i W_i$$
$$R_i = s_i H_p(P_i) + c_i W_I$$
$$c_{i+1} = H_s(\text{"CLSAG_round"} \,\|\, P \,\|\, C \,\|\, C' \,\|\, m \,\|\, L_i \,\|\, R_i)$$

In words: at each position `i`, two points are computed, `L_i` and `R_i`, and from them comes the challenge `c` of the next position. If the response `s_i` is the correct one for that position, the challenge obtained matches the published one.

- `s_i`: the response at position `i`. For the fake ones it is a random number.
- `c_i`: the challenge arriving at position `i`.
- `W_i`: the combined key of that position (previous step).
- `W_I`: the combined key image, to tie the signature to `I`.
- `L_i`, `R_i`: the two points that feed the hash; `L` checks the key and `R` checks the key image.
- `m`: the signed message; that way the signature is only valid for that transaction.

Signing. Choose a random `α` and set `L_π = αG`, `R_π = α·Hp(P_π)`. Compute `c_{π+1}` and walk around the ring with random `s_i` for the fake indices until reaching `π`. There, close the loop with:

$$s_\pi = \alpha - c_\pi \left(\mu_P x + \mu_C z\right)$$

> **In words:** at the real position, the signer picks the response that makes the circle close exactly. Only they can do it because the formula needs their secret key.

The signature is `(c_0, s_0 … s_{n-1}, D)`.

Verification. All the `c_i` are recomputed starting from `c_0`, and it is required that closing the ring yields `c_0` again. Each member costs one multi-exponentiation in two bases (`G` and `Hp(P_i)`).

Why it works. At `π`, `L_π = s_π G + c_π W_π = αG` and `R_π = s_π Hp(P_π) + c_π W_I = α Hp(P_π)`, because `W_I = w·Hp(P_π)`. That is, the same `w` appears in both bases: that is what ties the key image to the ring key and the zero commitment to the same index.

What the signature does not do

It does not hide the ring size or its members. Its anonymity is that of a set of 16 and it degrades if an attacker can rule out members by other means (chs. 7 and 14).

CHAPTER 7

Decoy selection

The ring's cryptography is perfect, but its real privacy depends on how the 15 decoys are chosen: if the real one stands out statistically, the ring hides nothing. This is the part of the protocol that has been corrected most often.

The age problem

People mostly spend recent outputs. If decoys were chosen uniformly over the whole chain, the youngest member of the ring would almost always be the real one. Academic analyses from 2017 showed that this heuristic ("the newest is the real one") was right for a majority of historical

rings.

The current solution: gamma distribution

The age of an output when it is spent is well approximated by a gamma distribution over the logarithm of the age in seconds, with shape 19.28 and rate 1.61. The resulting median is about 1.8 days. The wallet draws each decoy's age from that distribution, converts it into a block and picks an output at random within that block:

```
import math, random

SHAPE, RATE = 19.28, 1.61   # gamma parameters (in log-seconds)
BLOCK_TIME = 120            # seconds
UNLOCK = 10 * BLOCK_TIME    # an output is spendable after 10 blocks

def decoy_age_seconds():
    x = math.exp(random.gammavariate(SHAPE, 1 / RATE))
    # simplified: the real implementation handles recent values separately
    return x - UNLOCK if x > UNLOCK else random.uniform(0, BLOCK_TIME)

# 1) age -> block height (using the average block time)
# 2) block -> random output among that block's RingCT outputs
# 3) repeat without duplicates until there are 15 decoys, then insert the real one
```

The goal is that, to an observer, the real member is a draw from the same distribution as any decoy.

Consensus rules on rings

- The ring size is fixed: 16 since the August 2022 hard fork (11 since October 2019). All users use the same one, so there is no size that gives away different software.
- Each member must be at least 10 blocks old.
- Members are listed sorted by global index and without repeats.
- Decoys are always RingCT outputs (with a commitment), never old outputs with a plaintext amount.

What still fails

- **Outdated parameters.** The gamma was fitted with historical data. If real behaviour changes (for example, more immediate spending through automation), the real one stands out again.
- **Outputs already identified as spent.** If an analysis shows that an output is the real one in another ring, it stops being a valid decoy and the ring loses a member. With old minimum-size rings this chain effect was serious.
- **Dependence on external knowledge.** If an adversary knows that certain outputs in a ring are theirs or a known third party's, they eliminate them (ch. 14).

FCMP++ eliminates this problem at the root by replacing the set of 16 with the set of all outputs on the chain (ch. 15).

CHAPTER 8

RingCT: confidential amounts

RingCT lets nodes verify that a transaction neither creates nor destroys coins without seeing any amount. It relies on the homomorphic property of Pedersen commitments (ch. 2).

Commitment of each output

For output `k` with amount `a_k`, the sender chooses a blinding factor `y_k` and publishes the commitment in `outPk`:

$$C_k = y_k\,G + a_k\,H$$

In words: each output carries a commitment: a sealed envelope with its amount. Everyone sees the envelope, nobody sees the amount and the sender cannot change it.

The factor `y_k` is not freely random: it is derived from the output's shared secret (ch. 4), so that the receiver can rebuild it. The amount travels in 8 bytes encrypted by XOR with another derivation of the same secret, in the `ecdhInfo` field.

Pseudo-outputs: why they exist

Each input includes a new commitment `C'_j` to the same amount as the spent output, with another blinding factor:

$$C'_j = y'_j\,G + a_j\,H$$

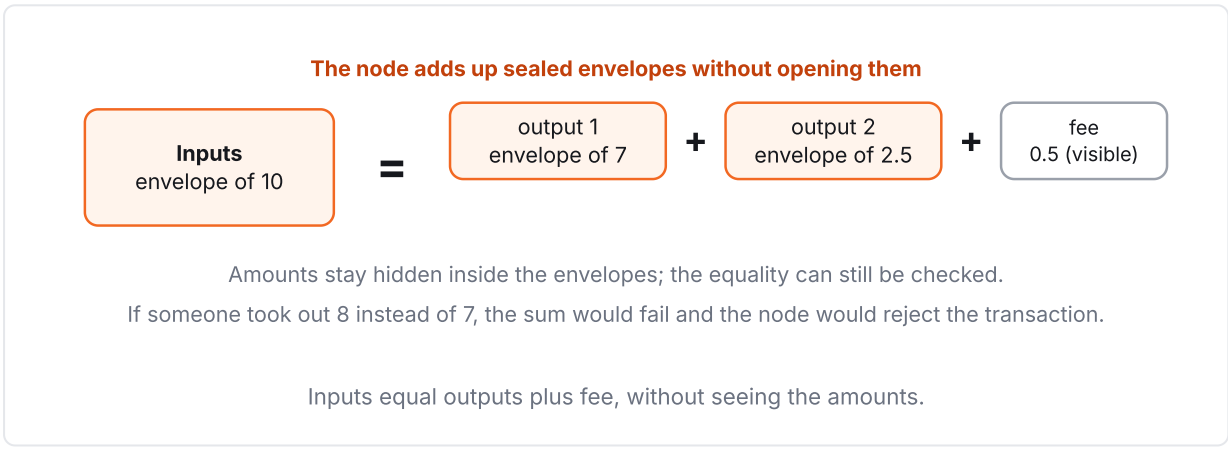
In words: each input re-seals the spent amount in a new envelope, with another blinding factor, so that this envelope cannot be matched with the original.

The CLSAG signature (ch. 6) proves, for the real ring member `π`, that `C_π - C'_j = zG`, that is, a commitment to zero. That shows that `C'_j` and the spent output have the same amount without saying which member it is.

Without pseudo-outputs the input commitments would have to be compared directly with the output ones, and the sum would give away which ring member is the real one. With them, the balance check happens outside the ring.

The balance equation

So that nobody creates money out of nothing, the node adds up the envelopes. What goes in has to equal what comes out plus the fee, which is the only thing visible in the clear. And it does so without opening any envelope.



The node checks, with the fee `f` in the clear:

KEY FORMULA.

$$\sum_j C'_j = \sum_k C_k + f\,H$$

In words: the sum of the input envelopes must equal the sum of the output envelopes plus one open envelope with the fee.

- `C'_j`: the envelope (pseudo-commitment) of input `j`.
- `C_k`: the envelope (commitment) of output `k`.
- `f`: the fee, written in the clear in the transaction.
- `H`: the point used to encode any amount; `f·H` is an envelope with no blinding factor.

Why it works. Commitments are linear: the sum of envelopes is the envelope of the sum. If the blinding factors also add up, the equality only holds when the input amounts are exactly the outputs plus the fee.

Example with small numbers. A coin of `10` goes in and `7` and `2.5` come out, with a fee of `0.5`. It adds up: `10 = 7 + 2.5 + 0.5`. If the sender tried to take out `8` instead of `7`, the sum of outputs would be `11` and the equality would fail without the node seeing any amount.

If it holds, `Σ a_j = Σ a_k + f` and `Σ y'_j = Σ y_k`. For the second to hold, the wallet freely chooses all the `y'_j` except the last, which it computes like this:

$$y'_{\text{last}} = \sum_{k} y_k - \sum_{j<\text{last}} y'_j$$

In words: the wallet freely chooses the blinding factors of all the inputs except the last, and adjusts that one so that the sums of the factors on both sides match.

The role of range proofs

All of the above is computed modulo `l`. Without further checks, a sender could use a "negative" amount (for example `l - 5`) in an output and the balance would still add up while creating coins out of nothing. Range proofs (ch. 9) guarantee that each `a_k` is in `[0, 2^64)`.

Special cases

- **Coinbase.** The miner reward amount is public and its commitment uses a fixed blinding factor `1`: `C = G + aH`. Anyone can recompute it.
- **Fee.** It is verified as a commitment with blinding factor `0` and amount `f`.
- **Transaction type.** On the current mainnet the RingCT type is `BulletproofPlus` (value 6 in the serialization), which combines CLSAG with Bulletproofs+.

CHAPTER 9

Range proofs: Bulletproofs+

A range proof shows that the amount hidden in a commitment `C = yG + aH` satisfies `0 ≤ a < 2^64`, without revealing it. With Bulletproofs+ this proof weighs about 640 bytes for a two-output transaction, compared with more than 12 kB for the original scheme.

Evolution

SCHEME	IN PRODUCTION SINCE	PROOF SIZE (2 OUTPUTS)	GROWTH WITH OUTPUTS
Borromean (original RingCT)	2017	on the order of 12 to 13 kB	Linear: about 6 kB per output
Bulletproofs	October 2018 hard fork	736 bytes	Logarithmic, with aggregation
Bulletproofs+	August 2022 hard fork	640 bytes	Logarithmic, with aggregation

The adoption of Bulletproofs in 2018 cut fees drastically. It is the most important size improvement in the protocol's history.

Idea of the proof

The range proof shows that an amount is an ordinary number, between zero and a maximum, without saying which. The idea is to write the amount in bits and prove that each bit is 0 or 1 and that together they rebuild the amount.

The amount is decomposed into bits `a_L ∈ {0,1}^64`. With `a_R = a_L - 1`, three conditions are equivalent to saying "`a` fits in 64 bits":

$$
\begin{aligned}
a_L \circ a_R &= 0 \quad \text{(each bit is 0 or 1)} \\
a_L - a_R - \mathbf{1} &= 0 \\
\langle a_L, \mathbf{2}^{64} \rangle &= a \quad \text{(the bits rebuild the amount)}
\end{aligned}
$$

In words: the three conditions say that each bit is worth 0 or 1 and that the bits, added with their weights of 1, 2, 4..., give exactly the amount. If they hold, the amount fits in 64 bits and is not negative.

They are combined into a single inner-product equality, which an *inner-product argument* proves in logarithmic time and size: in each round the vector is folded in half, and that is why the proof has `≈2·log2(64m)` points (`L_i`, `R_i`), with `m` outputs.

Bulletproofs+ uses a *weighted* inner-product argument that eliminates some of the original's intermediate commitments. The result is a proof three elements shorter (96 bytes) and somewhat faster verification.

Aggregation

A transaction with `m` outputs carries **a single** proof for all of them, padding `m` up to the next power of 2. The size grows 64 bytes each time the number of outputs doubles:

OUTPUTS	32-BYTE ELEMENTS	BULLETPROOFS+ SIZE
1	18	576 B
2	20	640 B
4	22	704 B
8	24	768 B
16 (consensus maximum)	26	832 B

That is why almost all transactions have two outputs (payment and change): a third forces a move to the next proof size and paying its verification cost (ch. 10).

Batch verification

Checking a proof is one big multi-exponentiation. For several proofs they are combined with random weights into a single multi-exponentiation (Straus- or Pippenger-type algorithms), so verifying a whole block costs considerably less than adding up the individual verifications. It is one of the reasons the initial sync is feasible.

CHAPTER 10

Anatomy of a transaction

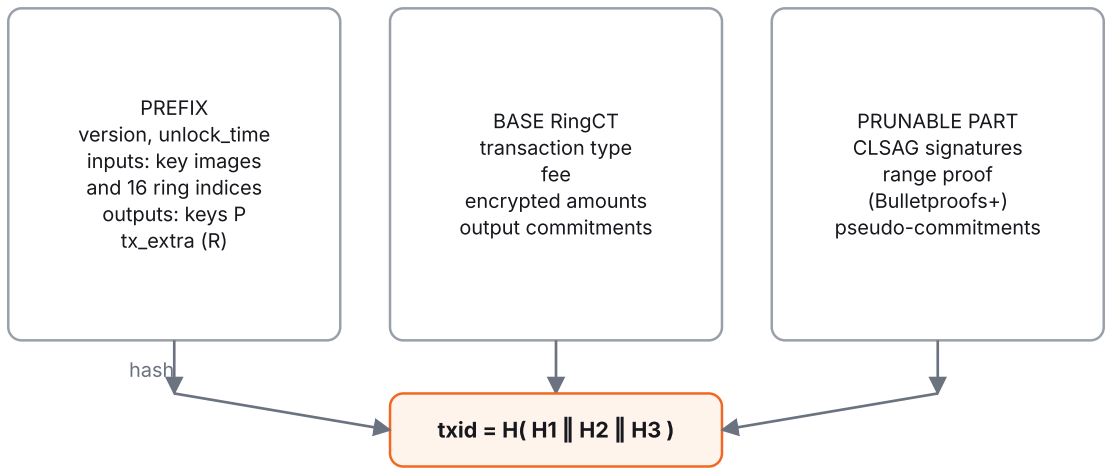
A mainnet transaction is divided into three blocks (prefix, RingCT base and prunable part), and that division explains how its hash is computed and how node pruning works.

Fields

BLOCK	FIELD	CONTENTS
Prefix	`version`, `unlock_time`	Version 2; `unlock_time` normally 0
Prefix	`vin[]`	Per input: `key_offsets` (16 global indices, relative) and `k_image` (32 B)
Prefix	`vout[]`	Per output: `amount = 0` (hidden), key `P` (32 B) and `view_tag` (1 B)
Prefix	`extra`	Public key `R`, additional keys for subaddresses, nonce with encrypted payment ID
RingCT base	`type`, `txnFee`	Type 6 (`BulletproofPlus`) and fee in the clear
RingCT base	`ecdhInfo[]`, `outPk[]`	Encrypted amount (8 B) and commitment `C_k` (32 B) per output
Prunable part	`bulletproofs_plus`	One aggregated range proof (ch. 9)
Prunable part	`CLSAGs[]`	One signature per input: `s_0..s_15`, `c_0`, `D` (ch. 6)
Prunable part	`pseudoOuts[]`	One commitment `C'_j` per input (ch. 8)

Transaction hash

The identifier is not the hash of the serialized bytes, but the hash of three hashes:



A node can discard the prunable part and keep the txid verifiable.

The three parts of a transaction and how the txid is obtained.

$$\text{txid} = H\big(\, H(\text{prefix}) \,\|\, H(\text{RingCT base}) \,\|\, H(\text{prunable part}) \,\big)$$

In words: the transaction identifier is the hash of three partial hashes, one for each part of the transaction.

That way, a node that discards the prunable part keeps its hash and the `txid` remains verifiable. Each input's signature covers a message that combines the hash of the prefix, that of the RingCT base and that of the range proofs, so nothing can be altered after signing.

Size of a typical transaction (2 inputs, 2 outputs)

Own estimate based on the fields above; the real figure varies by a few hundred bytes:

PART	APPROX. BYTES
2 inputs in the prefix (offsets + key image)	170
2 outputs in the prefix (key + view tag)	70
`extra`	45
RingCT base (type, fee, `ecdhInfo`, `outPk`)	85
2 CLSAG signatures (576 B each)	1150
Bulletproofs+ proof	650
2 `pseudoOuts`	65
Total	**≈ 2250**

The CLSAG signatures are roughly half of the transaction. That explains why FCMP++ (ch. 15), which replaces rings with a membership proof, changes the sizes so much.

Consensus rules worth remembering

- A transaction must have **at least 2 outputs** (if there is only one recipient, the change goes to a second output) and at most 16.
- There is no replace-by-fee (RBF): once in the mempool, the transaction is what it is.
- `unlock_time` prevents spending the outputs until a height or date. Normal outputs unlock after 10 blocks.

Fee and weight

The fee is `weight × fee per byte`. The weight is the size, plus a surcharge when the transaction has more than 2 outputs: Bulletproofs+ aggregation saves bytes but not verification time, and the surcharge (*clawback*) charges most of that saving. The base fee per byte is computed dynamically from the block reward and the median weight of recent blocks, and the user's priority multiplies it by a factor.

CHAPTER 11

Consensus and mining

Monero uses proof of work with RandomX, blocks of about 2 minutes, a perpetual tail emission of 0.6 XMR per block and an adaptive block size that replaces the fixed limit. Consensus offers no finality: only probabilistic.

RandomX

RandomX is a proof-of-work function designed so that a generic CPU is the most efficient hardware. Each attempt runs a randomly generated program on a virtual machine, with integer and floating-point operations, memory accesses and jumps, something hard to specialize in an ASIC.

- **Fast mode (miners):** uses a 2 GiB dataset in RAM.
- **Light mode (verifiers):** uses a 256 MiB cache and is slower per hash, but is enough to validate blocks.
- The key (*seed*) that defines the dataset changes every 2048 blocks, with a lag of 64 blocks so that everyone has time to prepare it.

RandomX has been in production since late 2019.

Difficulty

The target is one block every 120 seconds. Difficulty is recalculated on every block with a sliding window of 720 blocks: the timestamps are sorted, the extreme values (60 on each side) are discarded to limit manipulation, and the accumulated work is divided by the elapsed time.

Emission

The block reward has a decreasing part and a fixed minimum:

$$\text{reward} = \max\left(\frac{M - A}{2^{19}},\ 0.6\ \text{XMR}\right)$$

In words: the reward falls gradually with what has already been issued, but never below 0.6 XMR per block.

Where `M = 2^64 - 1` atomic units (1 XMR = 10^12 piconero) and `A` is what has already been issued. Since May 2022 the minimum rules: the **tail emission** amounts to about 157,800 XMR per year. The percentage inflation falls every year because the supply grows, and the goal is that miners always have an incentive and that lost coins are replenished.

Dynamic block size

There is no fixed limit. The maximum size is twice an effective median (between a short median of 100 blocks, a long one of 100,000 and a minimum of 300 kB). A block that exceeds the median `M_b` with weight `W` collects a reduced reward:

$$R_{\text{pen}} = R\left(1 - \left(\frac{W}{M_b} - 1\right)^2\right) \qquad M_b \le W \le 2M_b$$

In words: the bigger the block is relative to the median, the more reward the miner loses, and the loss grows quickly near twice the median.

The penalty is quadratic: it grows slowly at first and is severe near double. A miner only includes extra transactions if their fees exceed the lost reward. That makes fees rise only when there is sustained demand, and makes the median, and with it the limit, grow gradually.

Finality and reorganizations

Monero has no finality. Outputs are locked for 10 blocks (about 20 minutes) as protection against short reorganizations.

In 2025 that assumption was put to the test. The Qubic project directed hashrate to Monero with economic incentives and, according to The Block, in August it caused a six-block reorganization; whether it really reached 51% of the hashrate was a matter of debate. In September, according to Cointelegraph, there was an 18-block reorganization that exceeded the 10-block lock and reverted about 117 transactions. It is the largest in the network's history.

The security of a moderate-hashrate PoW depends on no actor being able to rent or attract more than half of it, and exchanges should ask for confirmations well above 10. The protocol's countermeasures (stricter chain-selection rules, checkpoints) are still under discussion, so it is worth checking the current state in the project's sources.

CHAPTER 12

P2P network and transport layer

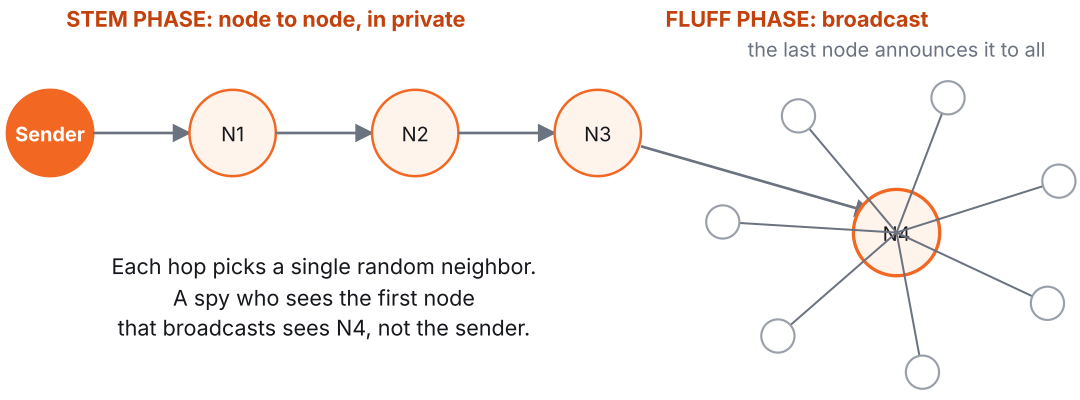
Cryptography protects what is inside the transaction, but the network can give away where it comes from. Monero uses a peer-to-peer (P2P) network, broadcasts transactions in a way that makes the origin hard to locate, and supports sending transactions through Tor or I2P.

Nodes and ports

The `monerod` program is the node. It talks to other nodes over the P2P port (18080 by default) and serves wallets over the RPC port (18081; 18089 if it is a restricted RPC for public use). A full node stores the entire chain, verifies all the consensus rules and does not need to trust anyone.

Transaction broadcast: Dandelion++

In a naive broadcast, the node that sees a transaction first is probably close to whoever created it. An eavesdropper with many nodes could link the transaction to the sender's IP.



Dandelion++: a private stem, then general broadcast.

Monero implements **Dandelion++** to prevent this, in two phases:

1. **Stem phase.** The transaction passes from one node to a single randomly chosen neighbour, and so on for a few hops. Nobody along the way knows whether the previous node is the origin or just a relay.
2. **Fluff phase.** At a random hop, the node broadcasts it to all its neighbours like a normal transaction.

The origin ends up far from the point where it becomes public. It is not an absolute guarantee against an adversary who controls many nodes.

Tor and I2P

`monerod` can send **only the transactions** (not the chain) through Tor or I2P, so not even your neighbours know your IP. Block syncing goes over the normal network, because it is a lot of traffic and contains no information about you. It is the recommended option if you care about hiding your IP when sending.

Mempool and blocks

- Transactions that are not yet in a block wait in the **mempool**. An unconfirmed transaction expires after 3 days.
- Blocks propagate in compact form (*fluffy blocks*): the header and the list of identifiers are sent, and the peer requests the transactions it is missing.
- There is no replace-by-fee; for it to confirm sooner, it has to be sent with high priority from the start.

Who sees what: your own node versus a remote node

A wallet needs data from a node. If you use someone else's (remote) node, that operator sees your IP and your wallet's queries, can reject or delay your transactions, and could lie about decoys or fees.

OPTION	PRIVACY	COST
Own full node	Maximum: nobody sees your queries	Hundreds of GB of disk and an initial sync
Own pruned node	Equally good, verifies the same	About a third of the space; stores only part of the prunable data
Trusted remote node (your own on another server)	Good if you connect over Tor or a VPN	Maintaining a server
Public remote node	The weakest: the operator sees your IP and your activity	None

CHAPTER 13

Wallets in practice

A Monero wallet does not "store coins": it stores your keys and walks the chain looking for which outputs are yours. That difference explains why syncing takes time, why a restore height is needed and why read-only wallets exist.

The scanning cycle

1. The wallet asks the node for blocks starting from the last height checked.
2. For each transaction it computes the shared secret with the view key and the transaction's `R` (ch. 4). *View tags* discard almost all outputs belonging to others without further computation.
3. The outputs it recognizes are stored with their amount, decrypted with the same secret.
4. With the spend key it computes the key image of each of its outputs (ch. 5) and, on seeing that key image in a later input, marks it as spent.

That is why a **read-only** wallet (with the view key but without the spend key) sees what you receive but not what you spend, unless the key images are imported into it from the full wallet. It is the basis of the offline signing flow: the connected machine prepares the transaction and the machine without a network signs it.

Backup and restore

- The **seed** (25 words or another format) is the only copy needed: both keys are derived from it.
- When restoring you have to give the **restore height**, that is, the block from which to start scanning. If you put one earlier than creation, you waste time; if you put a later one, you will not see old funds. With Polyseed the creation date is already inside the seed.
- The wallet files (keys encrypted with your password and scan cache) are just a convenience. They can be regenerated from the seed.

Spending: why the balance gets "locked"

Every new output, including change, takes 10 blocks (about 20 minutes) before it can be spent. If your wallet has a single large output and you make a payment, the change is locked and you cannot send another payment until it unlocks. Having several outputs of reasonable size avoids

that bottleneck.

When building a transaction the wallet chooses which outputs to spend, asks the node for the decoys (ch. 7), calculates the fee (ch. 10), signs and sends it. Spending many small outputs at once makes the transaction larger and more expensive.

Wallet variants

TYPE	WHO HOLDS THE KEYS	PRIVACY TRADE-OFF
Full (official GUI or CLI, Feather)	Your machine	Maximum, if you use your own node
Hardware (Ledger, Trezor)	The device holds the spend key; the view key is shared with the computer	Very good; spending is protected
Mobile with remote node (Cake Wallet and similar)	Your phone	Depends on the node you connect to (ch. 12)
Light wallet server	You hand your view key to a third party	The server sees everything you receive and spend
Read-only	Only the view key	Useful for accounting and auditing with no risk of spending

Multisig

Monero supports M-of-N signatures (for example 2 of 3), and to build them the participants exchange several rounds of data before they can sign. The feature is considered experimental and has to be enabled explicitly in the console wallet. Do not use it with important funds without understanding its limits.

CHAPTER 14

Attacks and the limits of privacy

Monero's privacy is rarely broken by attacking the cryptography; it is eroded by what surrounds it: small rings, counterparties who know too much, network metadata and user mistakes. Knowing these limits is part of understanding the protocol.

Catalogue of attacks

ATTACK	WHAT IT EXPLOITS	MITIGATION
Timing heuristic ("the newest is the real one")	Decoys chosen without imitating the real spending pattern	Gamma distribution (ch. 7); FCMP++ eliminates it (ch. 15)
Decoys discarded by chain effect	Outputs already known to be spent stop counting as decoys	Mandatory RingCT and minimum ring size; ring of 16
Flooding with own outputs	An attacker creates many outputs and knows which ones in a ring are theirs, and therefore not real	Larger ring size; FCMP++
Eve-Alice-Eve (EAE)	The counterparty knows the origin and destination of your funds and can narrow down which input you spent	Wait between receiving and spending; do not repeat patterns; FCMP++
Correlation with exchanges	Withdrawals and deposits with identity (KYC) linked by timing and approximate amount	Separate timing and amounts; use subaddresses
Network analysis	The IP your transaction comes from	Dandelion++, Tor or I2P, own node (ch. 12)
Malicious remote node	Serves you skewed decoys or measures your activity	Own node; check decoy distribution
Wallet fingerprints	Versions or settings that produce distinguishable transactions	Uniform consensus rules; keep the wallet updated

How much an attacker with their own outputs shrinks the ring

If each decoy has probability `p` of being an output controlled by the attacker (and therefore discardable), the effective ring size is approximately:

$$n_{\text{eff}} \approx 1 + (n - 1)(1 - p)$$

> **In words:** each decoy the attacker controls disappears from the ring, so as many plausible members remain as there are uncontrolled decoys, plus the real one.

With `n = 16` and `p = 0.5`, about 8.5 plausible members remain. Privacy falls without the attacker breaking anything: they only reduce the set you hide in. That is why increasing `n`, and ultimately using the whole chain, is a real improvement.

Mistakes users make

- Publishing an address next to your identity and receiving on it, or showing screenshots of payment receipts.

- Sharing the view key with third parties without thinking that they will see all your income forever.
- Reusing the same address with many contacts instead of subaddresses.
- Always spending immediately and with the same pattern, which makes timing correlation easier.
- Mixing Monero with services that demand identity without taking into account that the other party keeps what it sees.

What Monero does not promise

The chain hides who pays whom and how much, but it does not protect what you reveal outside it. It does not hide that a transaction existed, nor its size, nor the moment. And an adversary with access to your computer, your phone or your network connection has more information than anyone who only reads the chain.

CHAPTER 15

The future: FCMP++ and Carrot

FCMP++ replaces the 16-member ring with a proof that your output is in the set of all outputs on the chain, and Carrot renews the key and address scheme. As of today, November 2026, they are still in testing and have no confirmed activation date on mainnet.

Current status

According to Bitrue (12 October 2026), the 5 October hard fork took place on the *stressnet* test network at block 3,102,800 and did not affect the mainnet or balances. The same source says there is no confirmed date for the main hard fork and that a non-binding roadmap points to week 16 of 2027. Some low-quality page claims that FCMP++ is already active; recent, detailed sources say the opposite. Before relying on a date, check the project's official announcement.

The idea behind FCMP++

With rings, anonymity is that of 16 candidates. With FCMP++ the signer proves, without revealing which one, that their output belongs to the complete set of existing outputs, which the cited source puts at more than 150 million. Decoys are no longer chosen, so the attack surface of chapter 7 disappears.

To make it viable, the design combines three pieces:

- **Curve tree.** All outputs are organized in a Merkle-like tree whose levels alternate between two paired elliptic curves, so that membership proofs are small and do not depend on how big the chain is.
- **Zero-knowledge membership proof.** It proves that a path in the tree reaches an existing output, without saying which. It uses a variant of Bulletproofs.
- **Authorization and linking proof.** A re-randomized version of the output proves that you know its key and that the key image is the correct one. The key image stays: it is still needed to detect double spending.

What changes and what does not

ASPECT	TODAY (CLSAG, RING OF 16)	WITH FCMP++
Anonymity set	16 outputs	All outputs on the chain
Decoy selection	Needed and vulnerable to statistical attacks	Does not exist
Key images and double spending	Yes	Unchanged
Hidden amounts (Pedersen and Bulletproofs+)	Yes	Unchanged
Proof size per input	576 bytes	Larger, and grows little with the size of the chain
Nodes	Store outputs and key images	Also maintain the curve tree
Wallets	Ask the node for decoys	Ask for the tree branch of their output; with a remote node it is worth knowing what this reveals

I did not find reliable figures for size and verification cost to cite, so I do not include them.

Carrot

Carrot is the new address and key scheme designed to accompany FCMP++. Its specification separates capabilities that are mixed together today, such as generating addresses, viewing what was received, viewing the full balance and authorizing spends, so that each tool can be given only the key it needs. It also hardens the protocol against attacks in which someone tricks a receiver in order to identify their address.

For users, the wallet will have to be updated before the hard fork. The migration details (address format, existing seeds) should be confirmed in the official documentation as the date approaches.

Seraphis and Jamtis

Seraphis and Jamtis were an earlier and more ambitious proposal to redesign transactions and addresses. FCMP++ and Carrot pick up some of those ideas with a shorter implementation path. Whether Seraphis ever gets deployed remains uncertain.

CHAPTER 16

Lab: seeing the protocol at work

Four optional exercises, from easiest to hardest, to check with your own eyes what you have read. The first two require no programming; the third includes a complete, tested script.

Exercise 1: spin up a node on stagenet

Stagenet is a test network with the same rules as mainnet and coins without value, ideal for experimenting without risk. Download the official Monero binaries, verify their signatures and run:

```
monerod --stagenet --prune-blockchain --data-dir ~/monero-stagenet
```

Once it has synced (or already has blocks), ask the node for its status via RPC. On stagenet the RPC port is 38081 (on mainnet, 18081):

```
curl -s http://127.0.0.1:38081/json_rpc \
  -H 'Content-Type: application/json' \
  -d '{"jsonrpc":"2.0","id":"0","method":"get_info"}'
```

Look in the response for the height, the difficulty and the version of the active hard fork.

Exercise 2: dissect a real transaction

Pick a transaction identifier from a stagenet block explorer and request it from the node:

```
curl -s http://127.0.0.1:38081/get_transactions \
  -d '{"txs_hashes":["<TXID>"],"decode_as_json":true}'
```

In the JSON, the exact names vary a little between versions, but you should be able to check:

- how many members the ring of each input has (`key_offsets`) and where the key image is (`k_image`);
- that the outputs have `amount` equal to 0 and carry a one-time key and a *view tag*;
- that the RingCT type is 6 and that there is one commitment (`outPk`) per output and one `pseudoOut` per input;
- that there are at least two outputs, and how much each part weighs compared with the table in chapter 10.

Exercise 3: derive and scan a stealth output

This script implements from scratch, with curve arithmetic in pure Python, what is described in chapters 3, 4 and 8. It checks that sender and receiver reach the same shared secret, that the *view tag* matches, that the output's private key recovers the public key, and that Pedersen commitments add up. It needs `pip install pycryptodome` (for Keccak-256).

```
# stealth_demo.py: stealth addresses and Pedersen commitments over Ed25519.
# It is not a wallet: it does not encode addresses or sign transactions.
import secrets
from Crypto.Hash import keccak

p = 2**255 - 19
l = 2**252 + 27742317777372353535851937790883648493
d = -121665 * pow(121666, -1, p) % p
I = pow(2, (p - 1) // 4, p)  # square root of -1

def keccak256(data: bytes) -> bytes:
    return keccak.new(digest_bits=256, data=data).digest()

def Hs(*parts: bytes) -> int:            # hash to scalar
    return int.from_bytes(keccak256(b"".join(parts)), "little") % l

def varint(n: int) -> bytes:             # enough for n < 128
    return bytes([n])

# --- Curve arithmetic (extended coordinates, RFC 8032) ---
def add(P, Q):
    A = (P[1] - P[0]) * (Q[1] - Q[0]) % p
    B = (P[1] + P[0]) * (Q[1] + Q[0]) % p
    C = 2 * P[3] * Q[3] * d % p
    D = 2 * P[2] * Q[2] % p
    E, F, G_, H_ = B - A, D - C, D + C, B + A
    return (E * F % p, G_ * H_ % p, F * G_ % p, E * H_ % p)

ZERO = (0, 1, 1, 0)

def mul(k, P):                           # scalar multiplication
    R = ZERO
    while k:
        if k & 1:
            R = add(R, P)
        P = add(P, P)
```

```
        k >>= 1
    return R

def recover_x(y, sign):
    x2 = (y * y - 1) * pow(d * y * y + 1, -1, p) % p
    x = pow(x2, (p + 3) // 8, p)
    if (x * x - x2) % p:
        x = x * I % p
    if x & 1 != sign:
        x = p - x
    return x

def encode(P) -> bytes:
    zi = pow(P[2], -1, p)
    x, y = P[0] * zi % p, P[1] * zi % p
    return (y | ((x & 1) << 255)).to_bytes(32, "little")

Gy = 4 * pow(5, -1, p) % p
Gx = recover_x(Gy, 0)
G = (Gx, Gy, 1, Gx * Gy % p)

def rand_scalar():
    return secrets.randbelow(l - 1) + 1

# --- 1. Receiver's keys ---
b = rand_scalar()                                   # private spend key
a = Hs(b.to_bytes(32, "little"))                    # private view key
B, A = mul(b, G), mul(a, G)

# --- 2. The sender creates output k ---
k = 0
r = rand_scalar()
R = mul(r, G)                                       # transaction public key
D_sender = mul(8, mul(r, A))                        # derivation = 8·r·A
P = add(mul(Hs(encode(D_sender), varint(k)), G), B) # one-time key
tag = keccak256(b"view_tag" + encode(D_sender) + varint(k))[0]

# --- 3. The receiver scans ---
D_recv = mul(8, mul(a, R))                          # derivation = 8·a·R
assert encode(D_recv) == encode(D_sender), "Diffie-Hellman does not match"
tag_ok = keccak256(b"view_tag" + encode(D_recv) + varint(k))[0] == tag
P_check = add(mul(Hs(encode(D_recv), varint(k)), G), B)
mine = encode(P_check) == encode(P)
x = (Hs(encode(D_recv), varint(k)) + b) % l         # the output's private key
spendable = encode(mul(x, G)) == encode(P)

print("view tag matches  :", tag_ok)
print("the output is mine:", mine)
print("x*G == P          :", spendable)

# --- 4. Pedersen and its sum (toy H: NOT secure, only to see the linearity) ---
H = mul(7, G)
def commit(y, amount):
    return add(mul(y, G), mul(amount, H))

y1, a1, y2, a2 = rand_scalar(), 30, rand_scalar(), 12
lhs = add(commit(y1, a1), commit(y2, a2))
rhs = commit((y1 + y2) % l, a1 + a2)
print("C1 + C2 == C(sum) :", encode(lhs) == encode(rhs))
```

The expected result is four lines with `True`. Two warnings: the toy `H` (`7·G`) means anyone knows its discrete logarithm with respect to `G`, so that commitment is not binding; and Monero derives the real `H` by hash-to-point, something this script does not implement.

Exercise 4 (advanced): what breaks if you change something

Modify the script and reason about what you will observe before running it:

1. Change the receiver's index `k` to `1`. Is the output still theirs?
2. Scan with someone else's view key. What happens to the *view tag*? And to `P_check`?
3. Add a second output with another `k` and check that its one-time keys have no visible relationship.
4. Challenge: implement `Hp` (hash to point) and compute the key image `x·Hp(P)`. Validate your implementation against the test vectors in the official Monero repository.

CHAPTER 17

References and recommended reading

State of the protocol as of November 2026. Network figures change: always check the project's official sources before making decisions.

Sources for the current state (chapters 11 and 15)

These pages were consulted through web search, and they are the source for what concerns FCMP++ and the 2025 reorganizations:

- When Is the Next Monero Hard Fork? (Bitrue, 12 October 2026): status of FCMP++ and the stressnet date.
- Monero faces chain reorganization fears after Qubic says it controls 51% of hashrate (The Block): six-block reorganization, August 2025.
- Monero rallies 7% hours after 18-block reorg (Cointelegraph): 18-block reorganization, September 2025.

Recommended reading

They were not opened while writing; they are the standard references on the subject and it is worth checking that the links are still valid.

TOPIC	READING
Complete overview of the protocol	*Zero to Monero* (Alonso, Noether, Goodell), at getmonero.org
Origin of the design	*CryptoNote v2.0* (van Saberhagen, 2013)
CLSAG ring signatures	Goodell, Noether and RandomRun, *Concise Linkable Ring Signatures and Forgery Against Adversarial Keys*, IACR ePrint 2019/654
Range proofs	*Bulletproofs* (Bünz et al., ePrint 2017/1066) and *Bulletproofs+* (Chung et al., ePrint 2020/735)
Traceability analysis	Möser et al., *An Empirical Analysis of Traceability in the Monero Blockchain*, arXiv 1704.04299
Network	Fanti et al., *Dandelion++*, arXiv 1805.11060
PoW	The `tevador/RandomX` repository on GitHub, with its specification
Curve trees (basis of FCMP++)	Campanelli, Hall-Andersen and Kamp, *Curve Trees*, ePrint 2022/756
Reference code	The `monero-project/monero` repository on GitHub
Open research	Monero Research Lab, at getmonero.org

How to keep learning

Read *Zero to Monero* first with this guide beside you, to compare each concept with a longer explanation. After that, exercise 4 of the lab is the best way to check that you understand the key image and the ring signature.