MONERO

GUÍA DEL PROTOCOLO

Monero a fondo

Guía completa

Esta guía describe el protocolo de la mainnet de Monero (noviembre de 2026) con el detalle suficiente para entender cada una de sus piezas: primitivas, transacciones, consenso y red.

De los conceptos básicos a la criptografía

Contenido

INTRODUCCIÓN Y BASES

· Cómo usar esta guía 3

· Los conceptos básicos 3

· Criptografía desde cero 5

CAPÍTULOS

01 Visión general y modelo de amenazas 9

· Notación 11

02 Fundamentos criptográficos 11

03 Claves, direcciones y subdirecciones 14

04 Salidas sigilosas (stealth addresses) 16

05 Key images y prevención del doble gasto 19

06 Firmas de anillo: de MLSAG a CLSAG 21

07 Selección de señuelos 24

08 RingCT: montos confidenciales 26

09 Pruebas de rango: Bulletproofs+ 28

10 Anatomía de una transacción 30

11 Consenso y minería 34

12 Red P2P y capa de transporte 36

13 Wallets en la práctica 38

14 Ataques y límites de la privacidad 39

15 El futuro: FCMP++ y Carrot 41

16 Laboratorio: ver el protocolo funcionando 43

17 Referencias y lecturas recomendadas 46

INTRODUCCIÓN

Cómo usar esta guía

Público y prerrequisitos. No hace falta saber de criptografía ni de blockchain: la sección "Antes de empezar" explica desde cero los conceptos básicos y las matemáticas mínimas. Después la guía va subiendo de nivel hasta llegar a las fórmulas.

Estado del protocolo. Hoy la mainnet usa firmas CLSAG con anillos de 16 miembros y pruebas de rango Bulletproofs+. FCMP++ y Carrot están en pruebas y sin fecha de activación; se tratan en el capítulo 15.

Cómo leerla. Si partes de cero, empieza por "Antes de empezar" y consulta la tabla de notación, justo antes del capítulo 2, cuando llegues a las fórmulas. Cada capítulo abre con una frase que resume la idea; las fórmulas clave van marcadas, y cada una trae su explicación en palabras, el significado de cada símbolo y, cuando ayuda, un ejemplo con números pequeños. Los capítulos 2 a 10 son el núcleo criptográfico y deben leerse en orden. Los capítulos 11 a 14 (consenso, red, wallets, ataques) se pueden leer por separado. El capítulo 16 propone ejercicios de código.

ANTES DE EMPEZAR

Los conceptos básicos

Para entender Monero hacen falta cinco ideas previas: qué es una blockchain, qué son las claves y las firmas, qué es un hash, por qué Bitcoin no es privado y cómo se ve un pago de Monero a vista de pájaro. Si ya las dominas, salta al capítulo 1.

De un libro de cuentas central a uno compartido

En un banco, una sola entidad lleva el libro donde se anota quién tiene cuánto, y tú confías en ella. Una criptomoneda sustituye ese libro por uno **compartido**: miles de ordenadores (los *nodos*) guardan una copia idéntica y se ponen de acuerdo sobre su contenido sin que nadie mande.

- Las operaciones se agrupan en **bloques**. Cada bloque incluye la huella (hash) del anterior, así que los bloques forman una **cadena**: la *blockchain*.

- Cambiar una operación antigua obligaría a rehacer ese bloque y todos los siguientes, y la red lo detectaría.
- Los **mineros** compiten por añadir el siguiente bloque resolviendo un problema que cuesta mucha computación (la *prueba de trabajo*). El ganador cobra monedas nuevas más las comisiones. Reescribir la historia exigiría más potencia que todos los demás mineros juntos.

Claves y firmas

Cada usuario tiene dos números enlazados. La **clave privada** es secreta y sirve para autorizar pagos. La **clave pública** se deriva de ella y se puede compartir; funciona como el número de buzón donde cualquiera puede dejarte dinero.

Una **firma digital** demuestra que quien la hizo conoce la clave privada, sin revelarla, y queda atada a un mensaje concreto (la transacción). Si alguien cambia un solo dígito de la transacción, la firma deja de ser válida. La propiedad clave es que obtener la pública desde la privada es fácil, pero el camino inverso es inviable.

Hashes

Un **hash** convierte cualquier dato en una huella de tamaño fijo. Siempre da el mismo resultado para la misma entrada, cambia por completo si modificas un solo bit y no se puede invertir. Se usa para enlazar bloques, identificar transacciones, derivar claves y construir el problema de la minería.

Monedas como "salidas", no como saldos

Bitcoin y Monero no tienen cuentas con un saldo. Tienen **salidas**: cada una es una cantidad bloqueada para una clave concreta, como billetes sueltos. Para pagar, tu wallet gasta una o varias salidas enteras y crea otras nuevas: una para el destinatario y otra con el cambio para ti.

Una salida gastada no puede gastarse otra vez. Impedir ese **doble gasto** es el problema central de cualquier moneda digital, y en Monero se resuelve de una forma especial (cap. 5).

Por qué Bitcoin no es privado

En Bitcoin todo es público: qué dirección paga a cuál, cuánto, y qué salidas se gastan. No hay nombres, pero basta una compra con tu identidad en un solo punto para que empresas de análisis de cadena sigan el dinero hacia delante y hacia atrás. Ser seudónimo no es ser privado.

Eso tiene una segunda consecuencia, la **fungibilidad**: si el historial de una moneda es visible, una moneda "con mala fama" puede valer menos que otra idéntica. El dinero debería ser intercambiable unidad por unidad.

Un pago de Monero a vista de pájaro

1. El receptor te da su **dirección**, que es larga y reutilizable.
2. Tu wallet usa esa dirección para crear un **destino nuevo y único** para este pago, que solo el receptor sabe reconocer como suyo. En la cadena no aparece su dirección (*dirección sigilosa*, cap. 4).
3. Para gastar tus monedas, el wallet las mete en un grupo con otras 15 salidas ajenas de la cadena y firma de forma que se demuestra que *una* del grupo es tuya, sin decir cuál (*firma de anillo*, cap. 6). Una etiqueta única por salida (*key image*, cap. 5) evita que se gaste dos veces.
4. Los montos van en "sobres cerrados" (*compromisos*) que la red puede sumar sin abrir, para comprobar que no se crea dinero. Una prueba extra asegura que ningún sobre contiene un valor negativo (caps. 8 y 9).
5. Los mineros incluyen la transacción en un bloque y todos los nodos la verifican.

El resultado es privacidad en tres niveles a la vez: nadie sabe quién recibe, quién envía ni cuánto, y todo sigue siendo verificable por cualquiera.

ANTES DE EMPEZAR

Criptografía desde cero

Toda la privacidad de Monero sale de unas pocas ideas matemáticas, y todas se pueden entender con relojes, pintura y sobres. Aquí no se supone ningún conocimiento previo.

Aritmética de reloj

En un reloj de 12 horas, 9 + 5 no es 14 sino 2: al pasar de 12 se vuelve a empezar. Eso es la **aritmética modular**. La criptografía usa lo mismo con relojes gigantes, con números de unos 77 dígitos, de modo que cada operación da un resultado que parece aleatorio.

La clave es que algunas operaciones son fáciles en un sentido y prácticamente imposibles en el contrario. Ejemplo con un reloj de 23 horas: calcular `5^6` y quedarse con el resto da 8 enseguida. Averiguar *qué exponente* da 8 solo se puede probando uno por uno, y con relojes de 77 dígitos probar todos llevaría más tiempo que la edad del universo.

Curvas elípticas sin miedo

Una **curva elíptica** es un conjunto de puntos con una regla para "sumar" dos puntos y obtener un tercero. No hace falta saber geometría: basta con aceptar la regla, igual que aceptas que en un reloj 9 + 5 = 2.

Se elige un punto fijo y público, el **punto base** `G`. Sumar `G` consigo mismo `k` veces da otro punto, que se escribe `k·G`. Hay dos hechos fundamentales:

- **Calcular `k·G` es rápido**, aunque `k` tenga 77 dígitos: con unos 250 pasos basta.
- **Recuperar `k` a partir de `k·G` es inviable.** Es el *problema del logaritmo discreto*, y no se conoce atajo para la curva que usa Monero.

De ahí sale todo el esquema de claves: tu **clave privada** es un número `k` y tu **clave pública** es el punto `K = k·G`. Puedes publicar `K` sin riesgo.

Convención de esta guía: las minúsculas (`a`, `k`, `x`) son números; las mayúsculas (`A`, `K`, `P`) son puntos. Puedes sumar puntos y multiplicar un número por un punto, pero no multiplicar dos puntos. Monero usa una curva concreta llamada **Ed25519**.

Una propiedad será muy útil más adelante, la **linealidad**: `(a + b)·G = a·G + b·G`. Permite combinar valores secretos operando solo con sus versiones públicas.

Compartir un secreto en público (Diffie-Hellman)

La analogía clásica es la pintura. Alice y Bob acuerdan un color base, cada uno le mezcla un color secreto y se intercambian el resultado. Luego cada uno añade su propio secreto al mezclado del otro y ambos obtienen el mismo color final. Quien miró el intercambio no puede "des-mezclar" nada.

Con curvas es igual. Alice tiene `a` y publica `A = a·G`. Bob tiene `r` y publica `R = r·G`. Entonces:

FÓRMULA CLAVE.

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

En palabras: Bob multiplica la clave pública de Alice por su número secreto, Alice multiplica la de Bob por el suyo, y los dos llegan al mismo punto sin haberse revelado nada.

- `a` y `r`: los números secretos de Alice y de Bob. Nunca viajan por la red.
- `G`: el punto base de la curva. Lo conoce todo el mundo y es el punto de partida común.
- `A = a·G` y `R = r·G`: las claves públicas. Se pueden publicar porque, a partir de ellas, recuperar `a` o `r` es inviable.
- `r·A` y `a·R`: lo que calcula cada uno con su secreto y la clave pública del otro.

Por qué funciona. El orden de las multiplicaciones no importa: `r·(a·G)` y `a·(r·G)` son los dos `(a·r)·G`. Quien solo ve `A` y `R` tendría que deshacer una de las multiplicaciones, y eso es el problema del logaritmo discreto, para el que no se conoce solución eficiente.

Ejemplo con números pequeños. Con un reloj de 17 horas y `g = 3` (aquí «multiplicar» se escribe como potencia): Alice elige `a = 4` y publica `3^4 mod 17 = 13`. Bob elige `r = 6` y publica `3^6 mod 17 = 15`. Alice calcula `15^4 mod 17 = 16` y Bob calcula `13^6 mod 17 = 16`. Mismo número, y quien solo vio 13 y 15 no lo tuvo que conocer. Con 17 posibilidades se puede probar a mano; con los números de unos 76 dígitos de la curva real, no.

Bob calcula `r·A` con lo que Alice publicó, Alice calcula `a·R` con lo que Bob publicó, y ambos obtienen el mismo punto. Ese es su **secreto compartido**, y nadie más puede calcularlo. Monero lo usa para que el emisor y el receptor de un pago se pongan de acuerdo en una clave que solo ellos conocen (cap. 4).

Firmas: demostrar que conoces `k` sin decirlo

Una firma debe convencer a cualquiera de que conoces `k` tal que `K = k·G`, sin revelar `k`. El truco, en tres pasos:

1. **Compromiso:** eliges un número aleatorio `α` y publicas `α·G`.
2. **Desafío:** se calcula `c`, un número obtenido con un hash de todo lo relevante (incluido el mensaje). Nadie puede elegirlo a su favor.
3. **Respuesta:** publicas `s = α - c·k`.

Quien verifica comprueba una igualdad que solo se cumple si `s` se calculó con el `k` correcto:

FÓRMULA CLAVE.

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

> **En palabras:** quien verifica no necesita el secreto. Con la clave pública y los números de la firma comprueba que todo cuadra, y eso solo pudo conseguirlo quien conocía la clave privada.

- `k` y `K = k·G`: la clave privada y su clave pública.
- `α`: el aleatorio que tapa el secreto; sin él, `s` delataría `k`.
- `c`: el desafío, que sale de un hash del mensaje y de `α·G`; nadie puede elegirlo a su favor.
- `s = α - c·k`: la respuesta que se publica.

Ejemplo con números pequeños (tratando `G` como una unidad): con `k = 3` y `α = 7`, y un desafío `c = 2`, la respuesta es `s = 7 - 2·3 = 1`. El verificador, que solo conoce `K = 3G`, calcula `1·G + 2·K = 1G + 6G = 7G`, que es `α·G`, justo lo que se publicó al principio. Si alguien no conociera `k`, no podría fabricar una `s` que cuadre.

Como `s` está enmascarado por el aleatorio `α`, no revela `k`. Y como `c` depende del mensaje, la firma solo vale para esa transacción. Las firmas de anillo de Monero son una versión más elaborada de este mismo esquema.

Compromisos: el sobre cerrado

Un **compromiso** es un sobre cerrado: te fija a un valor sin revelarlo, y más tarde puedes abrirlo sin que nadie dude de lo que contiene. Un buen compromiso cumple dos cosas: **oculta** el valor y es **vinculante** (no puedes cambiarlo después).

Monero usa el de Pedersen: `C = y·G + a·H`, donde `a` es el monto, `y` es un número aleatorio que hace de papel opaco del sobre, y `H` es un segundo punto fijo. Por la linealidad, **sumar dos sobres da el sobre de la suma**, y eso permite comprobar que las entradas igualan las salidas sin abrir ninguno (cap. 8).

Pruebas de conocimiento cero

Una prueba de **conocimiento cero** convence de que algo es cierto sin enseñar por qué. Ya has visto una: la firma demuestra que conoces `k` sin mostrarlo. Monero usa otras dos: "mi salida es una de estas 16" (firma de anillo) y "este monto está entre 0 y 2^64" (prueba de rango).

Glosario rápido

TÉRMINO	EN UNA FRASE
Nodo	Ordenador que guarda la cadena y verifica todas las reglas
Minero	Quien propone bloques resolviendo la prueba de trabajo
Wallet	Programa que guarda tus claves, crea transacciones y escanea la cadena
Salida (output)	Cantidad de monedas bloqueada para una clave; lo que se gasta
Clave privada / pública	Número secreto `k` y su punto `k·G`, que se puede compartir
Clave de gasto / de vista	Dos claves de Monero: una para gastar, otra solo para ver lo recibido
Dirección sigilosa	Destino de un solo uso que se deriva de tu dirección para cada pago
Firma de anillo	Firma que demuestra que una de varias claves firmó, sin decir cuál
Key image	Etiqueta única por salida gastada que impide el doble gasto
Compromiso de Pedersen	Sobre cerrado para un monto, que se puede sumar sin abrir
Prueba de rango	Prueba de que un monto oculto no es negativo ni gigantesco
Mempool	Sala de espera de transacciones aun no incluidas en un bloque
Hard fork	Actualización del protocolo que todos los nodos deben adoptar a partir de cierto bloque
Piconero	Unidad mínima: 1 XMR = 10^12 piconero

CAPÍTULO 1

Visión general y modelo de amenazas

Monero oculta tres cosas por defecto en cada transacción (receptor, emisor y monto) y deja público todo lo demás. Entender qué queda visible es tan importante como entender qué se

oculta.

QUÉ SE OCULTA	MECANISMO	QUÉ QUEDA VISIBLE EN LA CADENA
Receptor	Direcciones sigilosas (cap. 4)	Una clave de un solo uso `P` por salida y la clave pública de la tx `R`
Emisor	Firma de anillo CLSAG (cap. 6)	Los 16 miembros del anillo y la key image (cap. 5)
Monto	Compromisos de Pedersen y Bulletproofs+ (cap. 8-9)	La comisión, el número de entradas y salidas, el tamaño

Fungibilidad como objetivo de diseño. Si cada moneda tiene un historial rastreable, unas valen más que otras. Monero busca que las salidas sean indistinguibles entre sí, y por eso la privacidad es obligatoria y no opcional.

Adversarios que conviene distinguir

1. **Observador pasivo de la cadena.** Tiene todo el historial público y aplica análisis estadístico a los anillos. Es el adversario contra el que se diseñó CLSAG y la selección de señuelos.
2. **Operador de nodo o espía de red.** Ve desde qué IP sale una transacción y, si eres cliente de un nodo remoto, qué salidas consultas. La cadena no te protege de esto.
3. **Contraparte o exchange.** Conoce la dirección y el momento de un pago concreto. Puede cruzar esos datos con su propia visión de la cadena.
4. **Atacante con salidas propias (EAE).** Si controla una fracción grande de las salidas usadas como señuelos, descarta anillos y reduce la ambigüedad (cap. 14).

Costes de la privacidad

Cada transacción pesa alrededor de 2 kB frente a unos 200 a 250 bytes de una transacción típica de Bitcoin. Los nodos no pueden saber qué salidas están gastadas, así que no existe un conjunto UTXO que se pueda podar. Y cada wallet debe escanear toda la cadena con su clave de vista para encontrar sus fondos.

REFERENCIA

Notación

Esta tabla fija los símbolos de las fórmulas que aparecen desde el capítulo 2.

SÍMBOLO	SIGNIFICADO
`G`	Punto base de Ed25519
`H`	Segundo generador, sin logaritmo discreto conocido respecto a `G` (se deriva por hash de `G`)
`l`	Orden del subgrupo primo: 2^252 + 27742317777372353535851937790883648493
minúscula (`a`, `x`)	Escalar módulo `l` (clave privada o aleatorio)
MAYÚSCULA (`A`, `P`)	Punto de la curva
`Hs(·)`	Keccak-256 reducido módulo `l` (hash a escalar)
`Hp(·)`	Hash a punto, multiplicado por 8 para caer en el subgrupo primo
`‖`	Concatenación de bytes

Monero usa el Keccak original (relleno previo a NIST), no SHA3-256.

CAPÍTULO 2

Fundamentos criptográficos

Este capítulo repasa con rigor lo visto en "criptografía desde cero": la privacidad de Monero descansa en la curva Ed25519, el hash Keccak, el intercambio Diffie-Hellman y los compromisos de Pedersen. Aquí están los detalles que importan al implementarlos.

Ed25519 y el cofactor

La curva tiene orden `8l`, con `l` primo. El cofactor 8 es fuente de errores sutiles, y Monero lo trata de tres formas:

- Las claves privadas son escalares reducidos módulo `l`.
- Las derivaciones Diffie-Hellman se multiplican por 8 para eliminar cualquier componente de torsión.
- Los nodos exigen que cada key image esté en el subgrupo primo, comprobando que `l·I` sea el elemento neutro. Sin esa comprobación, una key image podría modificarse con un punto de torsión y gastar dos veces la misma salida.

Hashes

`Hs(m)` es Keccak-256 de `m` reducido módulo `l`. `Hp(P)` aplica Keccak-256 a la codificación de `P`, lo mapea a un punto de la curva con una función de hash a punto propia de CryptoNote y lo multiplica por 8. El resultado es un punto cuyo logaritmo discreto respecto a `G` nadie conoce, y eso es lo que hace segura la key image (cap. 5).

Diffie-Hellman

Si `A = aG` y `R = rG`, ambos lados llegan al mismo secreto sin intercambiar claves privadas:

FÓRMULA CLAVE.

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

En palabras: es el mismo intercambio de antes con otras letras. El emisor (`r`) y el receptor (`a`) combinan cada uno su secreto con la clave pública del otro y llegan al mismo punto, que Monero usa como secreto compartido de cada pago.

El emisor lo calcula con `r` y la clave de vista pública del receptor; el receptor, con su clave de vista privada y la `R` publicada en la transacción. Es el núcleo de las salidas sigilosas.

Compromisos de Pedersen

Un compromiso al valor `a` con factor ciego `y` es:

FÓRMULA CLAVE.

$$C = yG + aH$$

En palabras: el compromiso es un sobre sellado que guarda el monto junto a un número aleatorio secreto: todos ven el sobre, nadie ve el monto y el autor no puede cambiarlo después.

- `a`: el monto que se quiere esconder.
- `y`: el factor ciego, un número aleatorio secreto que hace de papel opaco del sobre.
- `G`: el punto base público.
- `H`: un segundo punto público, elegido de forma que nadie sabe qué relación tiene con `G`; eso impide cambiar el monto después.
- `C`: el compromiso, un punto de la curva que se publica en la cadena.

Por qué funciona. Oculta porque `y·G` es un punto uniforme que tapa al monto. Es vinculante porque abrirlo con otro monto exigiría conocer la relación entre `G` y `H`. Y es lineal: sumar dos compromisos da el compromiso de la suma de los montos.

Ejemplo de juguete (números de un reloj de 101, solo para ver la suma; no es seguro): con `G = 2` y `H = 7`, el monto 5 con `y = 10` da `C = 10·2 + 5·7 = 55`, y el monto 3 con `y = 4` da `C = 4·2 + 3·7 = 29`. Sumados dan `84`, que es exactamente el compromiso al monto `8` con factor `14`: `14·2 + 8·7 = 84`. Así se comprueba que las cuentas cuadran sin abrir ningún sobre.

- **Oculta:** sin `y`, `C` no revela `a`, porque `yG` es un punto uniforme.
- **Vinculante:** nadie puede abrir `C` a dos valores distintos sin conocer `log_G(H)`.
- **Homomórfica:** `C(y1, a1) + C(y2, a2) = C(y1 + y2, a1 + a2)`. Los nodos verifican sumas de montos sin verlos.

Firmas tipo Schnorr y prueba de conocimiento

Todas las firmas de Monero (anillo, rango) son variantes de la prueba de conocimiento de un logaritmo discreto: compromiso, desafío derivado por hash (Fiat-Shamir) y respuesta. Si recuerdas la estructura `s = α - c·x`, ya tienes la mitad de CLSAG.

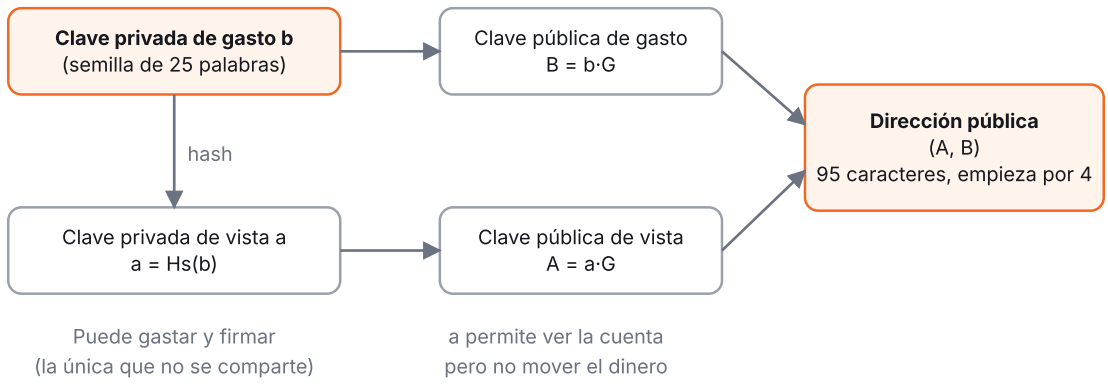
CAPÍTULO 3

Claves, direcciones y subdirecciones

Una cuenta Monero son dos pares de claves, y todo lo demás (semilla, dirección, subdirecciones) se deriva de ellas de forma determinista.

Jerarquía de claves

Una cuenta de Monero se apoya en dos claves privadas: una sirve para gastar y la otra para ver. La de vista se obtiene a partir de la de gasto, así que con una sola copia de seguridad se recupera todo. Quien tiene solo la clave de vista puede mirar la cuenta, pero no mover el dinero.



De la clave de gasto salen la de vista y la dirección pública.

$$b \in \mathbb{Z}_l \text{ (clave privada de gasto)} \qquad B = bG$$
$$a = H_s(b) \text{ (clave privada de vista)} \qquad A = aG$$

En palabras: la clave de vista sale de pasar la de gasto por una función hash, y cada clave privada tiene su clave pública, que es la que se comparte (`B` para gastar, `A` para ver).

- **Gasto (`b`):** permite firmar y gastar. Es la única clave que debe protegerse siempre.
- **Vista (`a`):** permite detectar fondos entrantes y descifrar montos, pero no gastar. Sirve para wallets de solo lectura y auditorías.

- La semilla mnemónica clásica son 25 palabras: 24 codifican los 256 bits de `b` y la última es un checksum. Existen formatos más nuevos, como Polyseed (16 palabras con fecha de creación), que algunos wallets ya soportan.

Dirección estándar

La dirección es la concatenación codificada en base58 (por bloques, no el base58 de Bitcoin):

```
byte de red (0x12 mainnet) ‖ B ‖ A ‖ checksum (4 primeros bytes de Keccak-256 de lo anterior)
```

Tiene 95 caracteres y empieza por `4`. Nótese que va primero la clave pública de gasto y después la de vista.

Subdirecciones

Una subdirección permite recibir en direcciones distintas sin que se puedan vincular entre sí en la cadena, y sin escanear con claves extra: sigue bastando la clave de vista `a`. Para el índice de cuenta `i` y de subdirección `j`:

$$
\begin{aligned}
m_{i,j} &= H_s(\text{"SubAddr\0"} \,\|\, a \,\|\, i \,\|\, j) \\
D_{i,j} &= B + m_{i,j}\, G \quad \text{(clave pública de gasto)} \\
C_{i,j} &= a \cdot D_{i,j} \quad \text{(clave pública de vista)}
\end{aligned}
$$

En palabras: cada subdirección se obtiene sumando a tu clave de gasto un desplazamiento calculado con tu clave de vista. Puedes crear tantas como quieras y, desde fuera, nadie puede saber que son de la misma persona.

La subdirección es el par `(C, D)`, empieza por `8` (byte de red 0x2a) y el emisor la trata como si fuera `(A, B)`, con una diferencia clave: publica `R = r·D` en lugar de `r·G`. El receptor recupera la clave de gasto de la salida, `P - Hs(a·R ‖ k)·G`, y la busca en su tabla de `D_{i,j}`. Por defecto el wallet precalcula 50 cuentas con 200 subdirecciones cada una.

Cuando una transacción paga a varios destinatarios y alguno es subdirección, `R` ya no sirve para todos y la transacción incluye una clave pública adicional por salida (`additional_tx_pub_keys`).

Payment IDs y direcciones integradas

Un payment ID era un identificador que el emisor adjuntaba para que un servicio distinguiera pagos a una misma dirección. Los de 32 bytes iban en claro y se eliminaron. Los de 8 bytes van cifrados con el secreto compartido y viven en las direcciones integradas (byte de red 0x13). Hoy

se desaconsejan a favor de subdirecciones, porque un payment ID es un identificador extra que degrada la privacidad.

Nota para el futuro: Carrot (cap. 15) cambia la jerarquía de claves y el formato de dirección, así que lo de este capítulo describe el esquema vigente en mainnet.

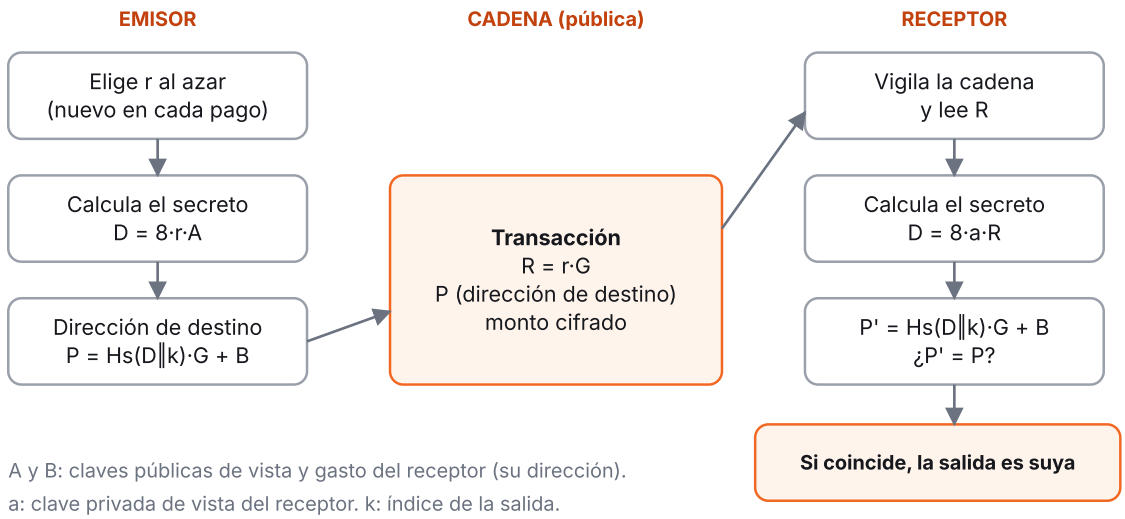
CAPÍTULO 4

Salidas sigilosas (stealth addresses)

Cada salida de una transacción va a una clave pública de un solo uso que solo el emisor y el receptor pueden relacionar con la dirección del receptor. Para un observador externo, dos pagos a la misma dirección son claves sin relación.

Derivación (lado del emisor)

Para cada pago el emisor genera un número aleatorio nuevo. Lo combina con la clave de vista pública del receptor para fabricar un secreto que solo ellos dos pueden calcular, y con ese secreto crea una dirección de destino de un solo uso para cada salida. En la transacción publica solo un punto auxiliar, que permite al receptor repetir el cálculo.



Cómo se crea y se reconoce una dirección de destino de un solo uso.

Para la salida de índice `k`, con una aleatoria `r` por transacción y la dirección `(A, B)` del receptor:

FÓRMULA CLAVE.

$$R = rG \quad \text{(clave pública de la tx, va en } \texttt{tx_extra}\text{)}$$
$$D = 8\,rA \quad \text{(derivación; el 8 elimina la torsión)}$$
$$P_k = H_s(D \,\|\, k)\,G + B$$

En palabras: el emisor publica un punto auxiliar, lo combina con la clave de vista del receptor para obtener un secreto compartido y lo usa para derivar la dirección de destino de cada salida.

- `r`: número aleatorio nuevo en cada transacción; es el secreto del emisor.
- `R = r·G`: punto auxiliar que se publica en la transacción (campo `tx_extra`).
- `A`, `B`: las claves públicas de vista y de gasto del receptor, que forman su dirección.
- `D = 8·r·A`: el secreto compartido (el Diffie-Hellman de antes). El `8` descarta una debilidad menor de la curva.
- `k`: el índice de la salida, para que dos salidas del mismo pago tengan direcciones distintas.
- `Hs(…)`: hash convertido en un número.
- `P_k`: la dirección de destino de un solo uso de la salida `k`: la clave `B` del receptor desplazada por algo que solo emisor y receptor conocen.

Por qué funciona. Un observador ve `R` y `P_k`, pero sin `a` o `r` no puede calcular `D`, así que no puede relacionar `P_k` con `B`. Cada salida parece una clave aleatoria sin vínculo con la dirección del receptor.

Recuperación (lado del receptor)

El receptor repite el cálculo desde su lado. Con su clave de vista privada y el punto auxiliar publicado obtiene el mismo secreto que el emisor, reconstruye la dirección de destino y comprueba si coincide con la de la salida. Si coincide, la salida es suya.

El receptor usa su clave de vista y la `R` de la transacción:

FÓRMULA CLAVE.

$$
\begin{aligned}
D &= 8\,aR = 8\,arG = 8\,rA \\
P'_k &= H_s(D \,\|\, k)\,G + B \qquad \text{si } P'_k = P_k \Rightarrow \text{la salida es mía} \\
x_k &= H_s(D \,\|\, k) + b \qquad \text{(clave privada para gastarla)}
\end{aligned}
$$

En palabras: el receptor llega al mismo secreto por el otro lado, reconstruye la dirección de destino, la compara con la de la salida y, si coincide, calcula la clave que le permite gastarla.

- `a·R`: lo mismo que `r·A`, porque `a·r·G = r·a·G`. Por eso emisor y receptor obtienen el mismo `D`.
- `P_k'`: la dirección de destino que el receptor calcula por su cuenta. Si es igual a la `P_k` de la salida, la salida es suya.
- `x_k`: la clave privada de la salida. Suma el desplazamiento secreto con `b`, la clave de gasto, que solo tiene el dueño: por eso quien solo tiene la clave de vista detecta el pago pero no puede gastarlo.

Solo quien tiene `b` conoce `x_k`, así que quien tenga únicamente la clave de vista puede detectar la salida y leer el monto, pero no gastarla.

Qué se cifra con el mismo secreto

Del mismo secreto compartido `H_s(D ‖ k)` se derivan, mediante hashes con etiquetas distintas, tanto el factor ciego `y` del compromiso como una máscara de 8 bytes que se aplica por XOR al monto. El receptor puede así reconstruir el compromiso y comprobar que coincide con el publicado (cap. 8).

Coste de escaneo y view tags

Escanear una transacción cuesta una multiplicación escalar `a·R`, y después un hash y una suma de puntos por salida. Sobre una cadena de decenas de millones de salidas, eso era lo que más tardaba en sincronizar un wallet.

Las **view tags** (desde el hard fork de agosto de 2022) añaden a cada salida un byte `Keccak("view_tag" ‖ D ‖ k)[0]`. El wallet compara ese byte primero y descarta de inmediato 255 de cada 256 salidas ajenas sin calcular `P_k'`. La sincronización mejora de forma notable (del orden de un 40%). El coste es que el byte filtra 8 bits sobre el secreto compartido, lo que se considera aceptable porque no permite a un tercero reconocer salidas.

Una clave `R` por transacción

La privacidad depende de que `r` sea nuevo en cada transacción. Si se reutilizara, todas las salidas de ese emisor hacia una misma dirección tendrían el mismo `D` y se podrían agrupar.

CAPÍTULO 5

Key images y prevención del doble gasto

La key image es una etiqueta determinista de cada salida gastada: no revela cuál es, pero si se gasta dos veces aparece repetida y la red lo detecta.

Construcción

La key image es una etiqueta que el dueño de una moneda calcula con su clave secreta. La misma moneda siempre produce la misma etiqueta, y la etiqueta no dice de qué moneda viene. Así, gastar dos veces la misma salida repite la etiqueta y la red lo detecta.

Si la salida es `P = xG` (con `x` la clave privada de la salida, cap. 4), su key image es:

FÓRMULA CLAVE.

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

En palabras: la key image es la clave privada de la salida aplicada sobre un punto que solo depende de esa salida. Siempre sale lo mismo para la misma moneda, y no se puede calcular sin el secreto.

- `x`: la clave privada de la salida (cap. 4). Es lo que solo conoce el dueño.
- `P`: la clave pública de la salida, con `P = x·G`.
- `Hp(P)`: un hash que convierte `P` en otro punto de la curva, del que nadie conoce su relación con `G`. Se usa un punto distinto de `G` para que `I` no sirva para deducir `P`.
- `I`: la key image que se publica al gastar.

Tres propiedades hacen que funcione:

1. **Determinista.** Para un mismo `x` solo hay un `I`. Cualquier intento de gastar `P` otra vez produce la misma `I`.
2. **Imposible de calcular sin `x`.** Quien ve `P` no puede obtener `I`, porque exigiría el logaritmo discreto de `P`. Quien ve `I` tampoco puede llegar a `P`.
3. **Verificable en cero conocimiento.** La firma de anillo (cap. 6) demuestra que existe un miembro del anillo con `P_π = xG` y que *el mismo* `x` produce `I = x·Hp(P_π)`. Es una prueba de igualdad de logaritmos discretos en dos bases, `G` y `Hp(P_π)`.

Sin la tercera propiedad el firmante podría inventar una `I` distinta cada vez y gastar la salida múltiples veces.

Qué verifica un nodo

Al recibir una transacción, para cada entrada el nodo comprueba:

- que `I` está en el subgrupo primo (`l·I` es el neutro),
- que `I` no figura en el conjunto de key images ya gastadas en la cadena ni en otra transacción del mempool,
- que la firma de anillo es válida para ese `I` (cap. 6).

El nodo mantiene en su base de datos (LMDB) la lista completa de key images gastadas. Esa lista solo crece.

Consecuencias de diseño

- **No hay UTXO set útil.** Como el nodo no sabe qué salida concreta corresponde a una key image, no puede marcarla como gastada ni podarla. Toda salida que haya existido sigue siendo candidata a señuelo.
- **La key image es pública y permanente.** Quien posea tu clave de gasto y de vista puede reconstruir tus key images y, por tanto, ver qué gastaste. Por eso la clave de vista sola no basta para ver gastos salientes, y por eso los wallets de solo lectura necesitan importar las key images para mostrar el saldo correcto.
- **En las entradas, las salidas del anillo se codifican como índices globales relativos** (`key_offsets`), que ahorran bytes al convertir la lista en diferencias entre índices ordenados.

CAPÍTULO 6

Firmas de anillo: de MLSAG a CLSAG

CLSAG demuestra, con un único bucle de desafíos, que el firmante conoce la clave de *una* salida del anillo y que la diferencia entre su compromiso y el pseudo-compromiso es un compromiso a cero, sin revelar cuál es.

Evolución del esquema

ESQUEMA	QUÉ APORTA	TAMAÑO DE FIRMA POR ENTRADA (ANILLO DE `N`)
Firma de anillo CryptoNote (2013)	Anonimato del emisor; key image para evitar doble gasto	Lineal en `n`, varios escalares por miembro
LSAG	Firma de anillo enlazable (linkable) con prueba compacta	`n + 1` escalares
MLSAG (RingCT, 2017)	Firma multicapa: demuestra clave y compromiso a la vez	`2n + 1` escalares (`32(2n+1)` bytes)
CLSAG (hard fork de octubre de 2020)	Fusiona las dos capas en una sola con coeficientes de agregación	`n + 2` elementos (`32(n+2)` bytes)

Con `n = 16`, una entrada pasa de 1056 bytes con MLSAG a 576 bytes con CLSAG, y se firma y verifica más rápido.

Datos de partida

Quien firma sabe cuál de las claves del anillo es la suya. Publica dos etiquetas calculadas con sus secretos: la key image, que impide gastar dos veces, y una imagen auxiliar que sirve para probar que los montos cuadran.

El anillo tiene claves `P_0 … P_{n-1}` y compromisos `C_0 … C_{n-1}`. El firmante conoce el índice real `π`, la clave `x` con `P_π = xG`, y el escalar `z = y_π - y'` tal que `C_π - C' = zG`, donde `C'` es el pseudo-compromiso de la entrada (cap. 8). Publica la key image y una imagen auxiliar:

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

En palabras: `I` es la key image de siempre. `D` es la misma idea aplicada al secreto `z` que une el monto gastado con su pseudo-compromiso; con ella la firma comprueba entradas y montos en una sola pasada.

Agregación de las dos capas

CLSAG debe comprobar dos cosas a la vez: que conoces una clave del anillo y que los montos cuadran. En lugar de hacerlo por separado, las mezcla en una sola comprobación con dos coeficientes que salen de un hash de todos los datos.

Dos coeficientes derivados por hash de todo el contexto (anillo, compromisos, `I`, `D`, `C'`) combinan la capa de claves y la de compromisos en una sola:

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

En palabras: los dos coeficientes `μ` mezclan la parte de las claves y la de los compromisos de cada miembro del anillo en una sola clave combinada `W`. No se pueden elegir a conveniencia porque dependen de todo el contexto.

En la posición real, `W_π = (μ_P x + μ_C z)·G`: una sola clave secreta `w = μ_P x + μ_C z` controla toda la fila.

El bucle de desafíos

La firma es una cadena de desafíos que se cierra sobre sí misma, como un círculo de fichas de dominó: cada posición del anillo produce un desafío que alimenta a la siguiente, y solo quien conoce la clave secreta puede cerrar el círculo. Para cualquiera que mire, todas las posiciones son indistinguibles.



El anillo de firmas como una cadena de desafíos cerrada.

Para cada índice `i` y con el mensaje `m` (hash del prefijo de la transacción y del resto de datos firmados):

FÓRMULA CLAVE.

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

En palabras: en cada posición `i` se calculan dos puntos, `L_i` y `R_i`, y de ellos sale el desafío `c` de la siguiente posición. Si la respuesta `s_i` es la correcta para esa posición, el desafío que se obtiene coincide con el publicado.

- `s_i`: la respuesta de la posición `i`. En las falsas es un número aleatorio.
- `c_i`: el desafío que llega a la posición `i`.
- `W_i`: la clave combinada de esa posición (paso anterior).
- `W_I`: la key image combinada, para vincular la firma con `I`.
- `L_i`, `R_i`: los dos puntos que alimentan el hash; `L` comprueba la clave y `R` comprueba la key image.
- `m`: el mensaje firmado; así la firma solo vale para esa transacción.

Firma. Elige un aleatorio `α` y fija `L_π = αG`, `R_π = α·Hp(P_π)`. Calcula `c_{π+1}` y recorre el anillo con `s_i` aleatorios para los índices falsos hasta llegar a `π`. Allí cierra el bucle con:

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

> **En palabras:** en la posición real, el firmante elige la respuesta que hace que el círculo cierre exactamente. Solo él puede hacerlo porque la fórmula necesita su clave secreta.

La firma es `(c_0, s_0 … s_{n-1}, D)`.

Verificación. Se recalculan todos los `c_i` desde `c_0` y se exige que al cerrar el anillo se obtenga de nuevo `c_0`. Cada miembro cuesta una multiexponenciación en dos bases (`G` y `Hp(P_i)`).

Por qué funciona. En `π`, `L_π = s_π G + c_π W_π = αG` y `R_π = s_π Hp(P_π) + c_π W_I = α Hp(P_π)`, porque `W_I = w·Hp(P_π)`. Es decir, la misma `w` aparece en las dos bases: eso es lo que ata la key image a la clave del anillo y el compromiso a cero al mismo índice.

Lo que la firma no hace

No oculta el tamaño del anillo ni sus miembros. Su anonimato es el de un conjunto de 16 y se degrada si un atacante puede descartar miembros por otros medios (cap. 7 y 14).

CAPÍTULO 7

Selección de señuelos

La criptografía del anillo es perfecta, pero su privacidad real depende de cómo se eligen los 15 señuelos: si el real se distingue estadísticamente, el anillo no oculta nada. Esta es la parte del protocolo que se ha corregido con más frecuencia.

El problema de la edad

La gente gasta sobre todo salidas recientes. Si los señuelos se eligieran de forma uniforme sobre toda la cadena, el miembro más joven del anillo sería casi siempre el real. Los análisis académicos de 2017 mostraron que esa heurística ("el más nuevo es el real") acertaba en una proporción mayoritaria de los anillos históricos.

La solución actual: distribución gamma

La edad de una salida al gastarse se aproxima bien con una distribución gamma sobre el logaritmo de la edad en segundos, con forma 19,28 y tasa 1,61. La mediana resultante es de unos 1,8 días. El wallet sortea la edad de cada señuelo con esa distribución, la convierte en un bloque y elige una salida al azar dentro de ese bloque:

```
import math, random

SHAPE, RATE = 19.28, 1.61   # parámetros de la gamma (en log-segundos)
BLOCK_TIME = 120            # segundos
UNLOCK = 10 * BLOCK_TIME    # una salida es gastable tras 10 bloques

def edad_decoy_segundos():
    x = math.exp(random.gammavariate(SHAPE, 1 / RATE))
    # simplificado: la implementación real trata aparte los valores recientes
    return x - UNLOCK if x > UNLOCK else random.uniform(0, BLOCK_TIME)

# 1) edad -> altura de bloque (usando el tiempo medio de bloque)
# 2) bloque -> salida al azar entre las salidas RingCT de ese bloque
# 3) se repite sin duplicados hasta tener 15 señuelos y se inserta el real
```

El objetivo es que, para un observador, el miembro real sea un sorteo de la misma distribución que cualquier señuelo.

Reglas de consenso sobre los anillos

- El tamaño del anillo es fijo: 16 desde el hard fork de agosto de 2022 (11 desde octubre de 2019). Todos los usuarios usan el mismo, de modo que no hay un tamaño que delate software distinto.
- Cada miembro debe tener al menos 10 bloques de antigüedad.
- Los miembros se listan ordenados por índice global y sin repetidos.
- Los señuelos son siempre salidas RingCT (con compromiso), nunca salidas antiguas con monto en claro.

Qué sigue fallando

- **Parámetros desfasados.** La gamma se ajustó con datos históricos. Si el comportamiento real cambia (por ejemplo, más gasto inmediato por automatización), el real vuelve a destacar.
- **Salidas ya identificadas como gastadas.** Si un análisis demuestra que una salida es la real en otro anillo, deja de ser un señuelo válido y el anillo pierde un miembro. Con anillos mínimos antiguos este efecto en cadena fue grave.
- **Dependencia del conocimiento externo.** Si un adversario sabe que ciertas salidas de un anillo son suyas o de un tercero conocido, las elimina (cap. 14).

FCMP++ elimina este problema de raíz al sustituir el conjunto de 16 por el conjunto de todas las salidas de la cadena (cap. 15).

CAPÍTULO 8

RingCT: montos confidenciales

RingCT permite que los nodos comprueben que una transacción no crea ni destruye monedas sin ver ningún monto. Se apoya en la propiedad homomórfica de los compromisos de Pedersen (cap. 2).

Compromiso de cada salida

Para la salida `k` de monto `a_k`, el emisor elige un factor ciego `y_k` y publica el compromiso en `outPk`:

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

En palabras: cada salida lleva un compromiso: un sobre sellado con su monto. Todos ven el sobre, nadie ve el monto y el emisor no puede cambiarlo.

El factor `y_k` no es aleatorio libre: se deriva del secreto compartido de la salida (cap. 4), para que el receptor lo reconstruya. El monto viaja en 8 bytes cifrados por XOR con otra derivación del mismo secreto, en el campo `ecdhInfo`.

Pseudo-outputs: por qué existen

Cada entrada incluye un compromiso nuevo `C'_j` al mismo monto que la salida gastada, con otro factor ciego:

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

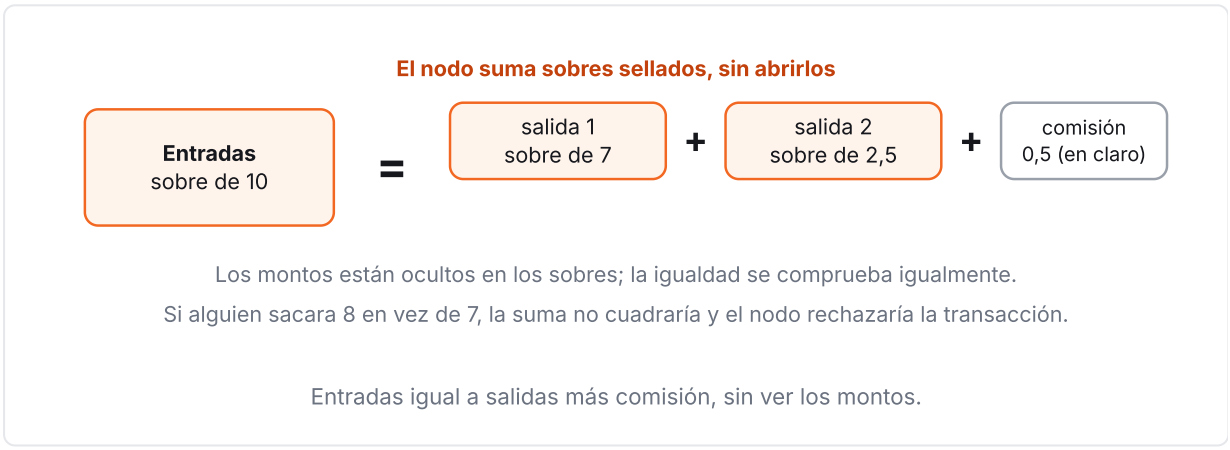
En palabras: cada entrada vuelve a sellar el monto gastado en un sobre nuevo, con otro factor ciego, para que ese sobre no se pueda emparejar con el original.

La firma CLSAG (cap. 6) prueba, para el miembro real `π` del anillo, que `C_π - C'_j = zG`, es decir, un compromiso a cero. Eso demuestra que `C'_j` y la salida gastada tienen el mismo monto sin decir cuál miembro es.

Sin pseudo-outputs habría que comparar directamente los compromisos de entrada con los de salida, y la suma delataría cuál miembro del anillo es el real. Con ellos, la verificación de balance ocurre fuera del anillo.

La ecuación de balance

Para que nadie cree dinero de la nada, el nodo suma los sobres. Lo que entra tiene que ser igual a lo que sale más la comisión, que es lo único que se ve en claro. Y lo hace sin abrir ningún sobre.



El nodo comprueba, con la comisión `f` en claro:

FÓRMULA CLAVE.

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

En palabras: la suma de los sobres de las entradas debe ser igual a la suma de los sobres de las salidas más un sobre abierto con la comisión.

- `C'_j`: el sobre (pseudo-compromiso) de la entrada `j`.
- `C_k`: el sobre (compromiso) de la salida `k`.
- `f`: la comisión, escrita en claro en la transacción.
- `H`: el punto con el que se codifica cualquier monto; `f·H` es un sobre sin factor ciego.

Por qué funciona. Los compromisos son lineales: la suma de sobres es el sobre de la suma. Si los factores ciegos también cuadran, la igualdad solo se cumple cuando los montos de entrada son exactamente los de salida más la comisión.

Ejemplo con números pequeños. Entra una moneda de `10` y salen `7` y `2,5`, con una comisión de `0,5`. Cuadra: `10 = 7 + 2,5 + 0,5`. Si el emisor intentara sacar `8` en lugar de `7`, la suma de salidas sería `11` y la igualdad fallaría sin que el nodo vea ningún monto.

Si se cumple, `Σ a_j = Σ a_k + f` y `Σ y'_j = Σ y_k`. Para que lo segundo se cumpla, el wallet elige libremente todos los `y'_j` salvo el último, que calcula así:

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

> **En palabras:** el wallet elige libremente los factores ciegos de todas las entradas menos la última, y esa la ajusta para que las sumas de factores de ambos lados coincidan.

El papel de las pruebas de rango

Todo lo anterior se calcula módulo `l`. Sin más controles, un emisor podría usar un monto "negativo" (por ejemplo `l - 5`) en una salida y que el balance siga cuadrando mientras crea monedas de la nada. Las pruebas de rango (cap. 9) garantizan que cada `a_k` esté en `[0, 2^64)`.

Casos especiales

- **Coinbase.** El monto de la recompensa del minero es público y su compromiso usa factor ciego fijo `1`: `C = G + aH`. Cualquiera puede recalcularlo.
- **Comisión.** Se verifica como un compromiso con factor ciego `0` y monto `f`.
- **Tipo de transacción.** En la mainnet actual el tipo RingCT es `BulletproofPlus` (valor 6 en la serialización), que combina CLSAG con Bulletproofs+.

CAPÍTULO 9

Pruebas de rango: Bulletproofs+

Una prueba de rango demuestra que el monto oculto en un compromiso `C = yG + aH` cumple `0 ≤ a < 2^64`, sin revelarlo. Con Bulletproofs+ esa prueba pesa unos 640 bytes para una transacción de dos salidas, frente a más de 12 kB del esquema original.

Evolución

ESQUEMA	EN PRODUCCIÓN DESDE	TAMAÑO DE LA PRUEBA (2 SALIDAS)	CRECIMIENTO CON LAS SALIDAS
Borromean (RingCT original)	2017	del orden de 12 a 13 kB	Lineal: unos 6 kB por salida
Bulletproofs	Hard fork de octubre de 2018	736 bytes	Logarítmico, con agregación
Bulletproofs+	Hard fork de agosto de 2022	640 bytes	Logarítmico, con agregación

La adopción de Bulletproofs en 2018 redujo las comisiones de forma drástica. Es la mejora de tamaño más importante de la historia del protocolo.

Idea de la prueba

La prueba de rango demuestra que un monto es un número normal, entre cero y un máximo, sin decir cuál. La idea es escribir el monto en bits y probar que cada bit es 0 o 1 y que juntos reconstruyen el monto.

Se descompone el monto en bits `a_L ∈ {0,1}^64`. Con `a_R = a_L - 1`, tres condiciones son equivalentes a decir "`a` cabe en 64 bits":

$$
\begin{aligned}
a_L \circ a_R &= 0 \quad (\text{cada bit es 0 o 1}) \\
a_L - a_R - \mathbf{1} &= 0 \\
\langle a_L, \mathbf{2}^{64} \rangle &= a \quad (\text{los bits reconstruyen el monto})
\end{aligned}
$$

En palabras: las tres condiciones dicen que cada bit vale 0 o 1 y que los bits, sumados con sus pesos de 1, 2, 4..., dan exactamente el monto. Si se cumplen, el monto cabe en 64 bits y no es negativo.

Se combinan en una sola igualdad de producto interno, que un *argumento de producto interno* demuestra en tiempo y tamaño logarítmicos: en cada ronda se pliega el vector a la mitad, y por eso la prueba tiene `≈2·log2(64m)` puntos (`L_i`, `R_i`), con `m` salidas.

Bulletproofs+ usa un argumento de producto interno *ponderado* que elimina algunos compromisos intermedios del original. El resultado es una prueba tres elementos más corta (96 bytes) y una verificación algo más rápida.

Agregación

Una transacción con `m` salidas lleva **una sola** prueba para todas, rellenando `m` hasta la potencia de 2 siguiente. El tamaño crece 64 bytes cada vez que se duplica el número de salidas:

SALIDAS	ELEMENTOS DE 32 BYTES	TAMAÑO BULLETPROOFS+
1	18	576 B
2	20	640 B
4	22	704 B
8	24	768 B
16 (máximo por consenso)	26	832 B

Por eso casi todas las transacciones tienen dos salidas (pago y cambio): una tercera obliga a pasar al siguiente tamaño de prueba y a pagar su coste de verificación (cap. 10).

Verificación por lotes

La comprobación de una prueba es una gran multiexponenciación. Para varias pruebas se combinan con pesos aleatorios en una sola multiexponenciación (algoritmos tipo Straus o Pippenger), con lo que verificar un bloque entero cuesta bastante menos que sumar las verificaciones individuales. Es una de las razones por las que la sincronización inicial es viable.

CAPÍTULO 10

Anatomía de una transacción

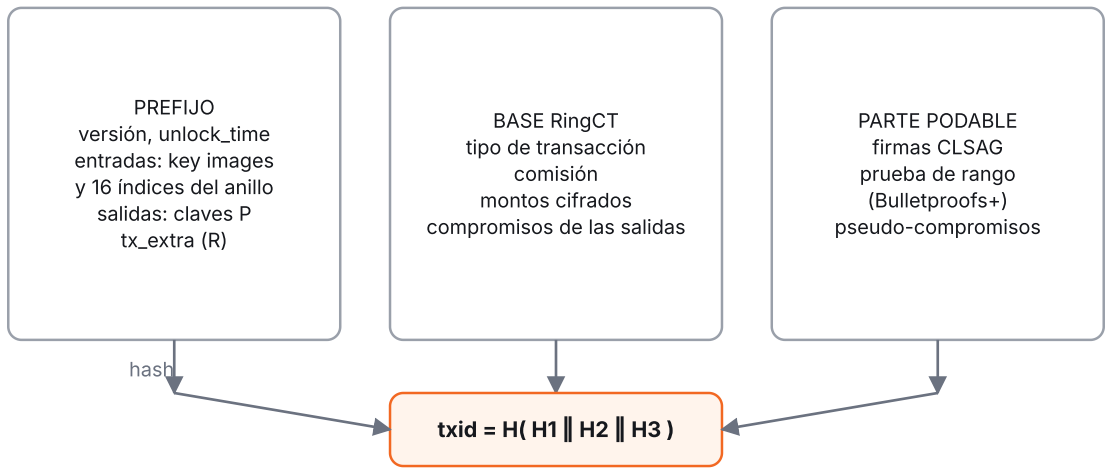
Una transacción de mainnet se divide en tres bloques (prefijo, base RingCT y parte podable), y esa división explica cómo se calcula su hash y cómo funciona la poda de nodos.

Campos

BLOQUE	CAMPO	CONTENIDO
Prefijo	`version`, `unlock_time`	Versión 2; `unlock_time` normalmente 0
Prefijo	`vin[]`	Por entrada: `key_offsets` (16 índices globales, relativos) y `k_image` (32 B)
Prefijo	`vout[]`	Por salida: `amount = 0` (oculto), clave `P` (32 B) y `view_tag` (1 B)
Prefijo	`extra`	Clave pública `R`, claves adicionales para subdirecciones, nonce con payment ID cifrado
Base RingCT	`type`, `txnFee`	Tipo 6 (`BulletproofPlus`) y comisión en claro
Base RingCT	`ecdhInfo[]`, `outPk[]`	Monto cifrado (8 B) y compromiso `C_k` (32 B) por salida
Parte podable	`bulletproofs_plus`	Una prueba de rango agregada (cap. 9)
Parte podable	`CLSAGs[]`	Una firma por entrada: `s_0..s_15`, `c_0`, `D` (cap. 6)
Parte podable	`pseudoOuts[]`	Un compromiso `C'_j` por entrada (cap. 8)

Hash de la transacción

El identificador no es el hash de los bytes serializados, sino el hash de tres hashes:



Un nodo puede descartar la parte podable y conservar el txid verificable.

Las tres partes de una transacción y cómo se obtiene el txid.

$$\text{txid} = H\big(\, H(\text{prefijo}) \,\|\, H(\text{base RingCT}) \,\|\, H(\text{parte podable}) \,\big)$$

En palabras: el identificador de la transacción es el hash de tres hashes parciales, uno por cada parte de la transacción.

Así, un nodo que elimina la parte podable conserva su hash y el `txid` sigue siendo verificable. La firma de cada entrada cubre un mensaje que combina el hash del prefijo, el de la base RingCT y el de las pruebas de rango, de modo que no se puede alterar nada después de firmar.

Tamaño de una transacción típica (2 entradas, 2 salidas)

Estimación propia a partir de los campos de arriba; la cifra real varía unos cientos de bytes:

PARTE	BYTES APROX.
2 entradas en el prefijo (offsets + key image)	170
2 salidas en el prefijo (clave + view tag)	70
`extra`	45
Base RingCT (tipo, comisión, `ecdhInfo`, `outPk`)	85
2 firmas CLSAG (576 B cada una)	1150
Prueba Bulletproofs+	650
2 `pseudoOuts`	65
Total	**≈ 2250**

Las firmas CLSAG son aproximadamente la mitad de la transacción. Eso explica por qué FCMP++ (cap. 15), que sustituye los anillos por una prueba de pertenencia, cambia tanto los tamaños.

Reglas de consenso que conviene recordar

- Una transacción debe tener **al menos 2 salidas** (si solo hay un destinatario, el cambio va a una segunda salida) y a lo sumo 16.
- No existe reemplazo por comisión (RBF): una vez en el mempool, la transacción es la que es.
- `unlock_time` impide gastar las salidas hasta una altura o fecha. Las salidas normales se desbloquean tras 10 bloques.

Comisión y peso

La comisión es `peso × comisión por byte`. El peso es el tamaño, más un recargo cuando la transacción tiene más de 2 salidas: la agregación de Bulletproofs+ ahorra bytes pero no tiempo de verificación, y el recargo (*clawback*) cobra la mayor parte de ese ahorro. La comisión por byte base se calcula de forma dinámica a partir de la recompensa de bloque y de la mediana de pesos de los bloques recientes, y la prioridad del usuario la multiplica por un factor.

CAPÍTULO 11

Consenso y minería

Monero usa prueba de trabajo con RandomX, bloques de unos 2 minutos, una emisión de cola perpetua de 0,6 XMR por bloque y un tamaño de bloque adaptativo que sustituye al límite fijo. El consenso no ofrece finalidad: solo probabilística.

RandomX

RandomX es una función de prueba de trabajo pensada para que una CPU genérica sea el hardware más eficiente. Cada intento ejecuta un programa generado al azar en una máquina virtual, con operaciones enteras y de punto flotante, accesos a memoria y saltos, algo difícil de especializar en un ASIC.

- **Modo rápido (mineros):** usa un dataset de 2 GiB en RAM.
- **Modo ligero (verificadores):** usa una caché de 256 MiB y es más lento por hash, pero basta para validar bloques.
- La clave (*seed*) que define el dataset cambia cada 2048 bloques, con un desfase de 64 bloques para que todos tengan tiempo de prepararla.

RandomX está en producción desde finales de 2019.

Dificultad

El objetivo es un bloque cada 120 segundos. La dificultad se recalcula en cada bloque con una ventana deslizante de 720 bloques: se ordenan las marcas de tiempo, se descartan los valores extremos (60 de cada lado) para limitar la manipulación y se divide el trabajo acumulado entre el tiempo transcurrido.

Emisión

La recompensa por bloque tiene una parte decreciente y un mínimo fijo:

$$\text{recompensa} = \max\left(\frac{M - A}{2^{19}},\ 0{,}6\ \text{XMR}\right)$$

En palabras: la recompensa baja poco a poco con lo ya emitido, pero nunca por debajo de 0,6 XMR por bloque.

Donde `M = 2^64 - 1` unidades atómicas (1 XMR = 10^12 piconero) y `A` es lo ya emitido. Desde mayo de 2022 manda el mínimo: la **emisión de cola** equivale a unos 157.800 XMR por año. La inflación porcentual disminuye cada año porque la oferta crece, y el objetivo es que los mineros siempre tengan incentivo y que las monedas perdidas se repongan.

Tamaño de bloque dinámico

No hay un límite fijo. El tamaño máximo es el doble de una mediana efectiva (entre una mediana corta de 100 bloques, una larga de 100.000 y un mínimo de 300 kB). Un bloque que supera la mediana `M_b` con peso `W` cobra una recompensa reducida:

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

En palabras: cuanto más grande es el bloque respecto a la mediana, más recompensa pierde el minero, y la pérdida crece rápido cerca del doble de la mediana.

La penalización es cuadrática: crece poco al principio y es severa cerca del doble. Un minero solo incluye transacciones extra si sus comisiones superan la pérdida de recompensa. Eso hace que las comisiones suban solo cuando hay demanda sostenida, y que la mediana, y con ella el límite, crezca poco a poco.

Finalidad y reorganizaciones

Monero no tiene finalidad. Las salidas se bloquean 10 bloques (unos 20 minutos) como protección frente a reorganizaciones cortas.

En 2025 esa hipótesis se puso a prueba. El proyecto Qubic dirigió hashrate a Monero con incentivos económicos y, según The Block, en agosto provocó una reorganización de seis bloques; que alcanzara de verdad el 51% del hashrate es algo que se discutió. En septiembre, según Cointelegraph, hubo una reorganización de 18 bloques que superó los 10 de bloqueo y revirtió unas 117 transacciones. Es la mayor de la historia de la red.

La seguridad de una PoW de hashrate moderado depende de que ningún actor pueda alquilar o atraer más de la mitad, y los exchanges deberían pedir confirmaciones muy por encima de 10. Las contramedidas del protocolo (reglas de selección de cadena más estrictas, puntos de control) siguen en discusión, así que conviene consultar el estado actual en las fuentes del proyecto.

CAPÍTULO 12

Red P2P y capa de transporte

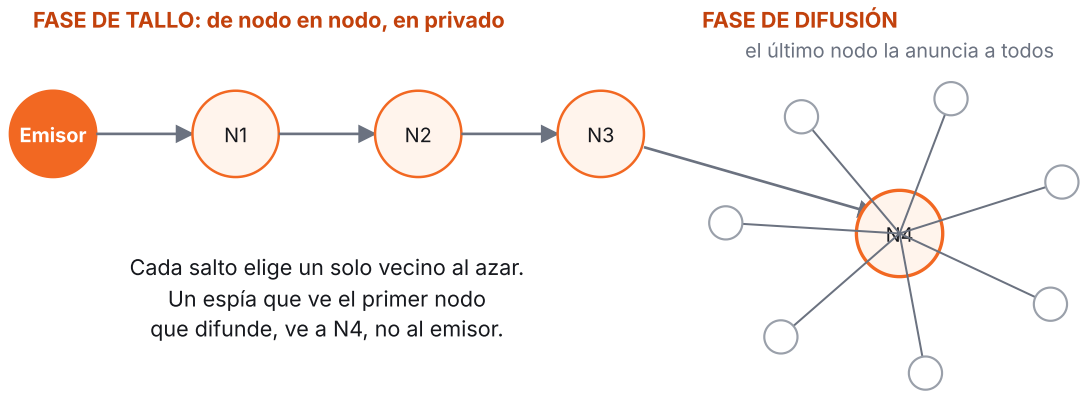
La criptografía protege lo que va dentro de la transacción, pero la red puede delatar de dónde sale. Monero usa una red entre iguales (P2P), difunde las transacciones de forma que cuesta localizar al origen, y admite enviar transacciones a través de Tor o I2P.

Nodos y puertos

El programa `monerod` es el nodo. Habla con otros nodos por el puerto P2P (18080 por defecto) y atiende a los wallets por el puerto RPC (18081; 18089 si es un RPC restringido para uso público). Un nodo completo guarda la cadena entera, verifica todas las reglas del consenso y no necesita confiar en nadie.

Difusión de transacciones: Dandelion++

En una difusión ingenua, el nodo que ve primero una transacción probablemente está cerca de quien la creó. Un espía con muchos nodos podría vincular la transacción con la IP del emisor.



Dandelion++: un tallo privado y después la difusión general.

Monero implementa **Dandelion++** para evitarlo, en dos fases:

1. **Fase de tallo (*stem*).** La transacción pasa de un nodo a un solo vecino elegido al azar, y así durante unos saltos. Nadie en el camino sabe si el anterior es el origen o un reenviador.
2. **Fase de difusión (*fluff*).** En un salto aleatorio, el nodo la difunde a todos sus vecinos como una transacción normal.

El origen queda lejos del punto desde el que se hace pública. No es una garantía absoluta contra un adversario que controle muchos nodos.

Tor e I2P

`monerod` puede enviar **solo las transacciones** (no la cadena) a través de Tor o I2P, de modo que ni siquiera tus vecinos conocen tu IP. La sincronización de bloques va por la red normal, porque es mucho tráfico y no contiene información tuya. Es la opción recomendada si te importa ocultar tu IP al emitir.

Mempool y bloques

- Las transacciones que aún no están en un bloque esperan en el **mempool**. Una transacción sin confirmar caduca a los 3 días.
- Los bloques se propagan en forma compacta (*fluffy blocks*): se envía la cabecera y la lista de identificadores, y el par pide las transacciones que le falten.
- No hay reemplazo por comisión; para que se confirme antes, hay que enviarla con prioridad alta desde el principio.

Quién ve qué: nodo propio frente a nodo remoto

Un wallet necesita datos de un nodo. Si usas un nodo ajeno (remoto), ese operador ve tu IP y las consultas de tu wallet, puede rechazar o retrasar tus transacciones, y podría mentir sobre señuelos o comisiones.

OPCIÓN	PRIVACIDAD	COSTE
Nodo propio completo	Máxima: nadie ve tus consultas	Cientos de GB de disco y sincronización inicial
Nodo propio podado	Igual de buena, verifica igual	Alrededor de un tercio del espacio; guarda solo parte de los datos podables
Nodo remoto de confianza (el tuyo en otro servidor)	Buena si te conectas por Tor o VPN	Mantener un servidor
Nodo remoto público	La más débil: el operador ve tu IP y tu actividad	Ninguno

CAPÍTULO 13

Wallets en la práctica

Un wallet de Monero no "guarda monedas": guarda tus claves y recorre la cadena buscando qué salidas son tuyas. Esa diferencia explica por qué sincronizar tarda, por qué hace falta una altura de restauración y por qué existen los wallets de solo lectura.

El ciclo de escaneo

1. El wallet pide bloques al nodo desde la última altura revisada.
2. Para cada transacción calcula el secreto compartido con la clave de vista y la `R` de la transacción (cap. 4). Las *view tags* descartan casi todas las salidas ajenas sin más cálculo.
3. Las salidas que reconoce las guarda con su monto, descifrado con el mismo secreto.
4. Con la clave de gasto calcula la key image de cada salida suya (cap. 5) y, al ver esa key image en una entrada posterior, la marca como gastada.

Por eso un wallet de **solo lectura** (con la clave de vista pero sin la de gasto) ve lo que recibes pero no lo que gastas, a menos que se le importen las key images desde el wallet completo. Es la base del flujo de firma offline: el equipo conectado prepara la transacción y el equipo sin red la firma.

Copia de seguridad y restauración

- La **semilla** (25 palabras u otro formato) es la única copia necesaria: de ella se derivan las dos claves.
- Al restaurar hay que indicar la **altura de restauración**, es decir, el bloque desde el que empezar a escanear. Si pones una anterior a la creación, pierdes tiempo; si pones una posterior, no verás fondos antiguos. Con Polyseed la fecha de creación ya va dentro de la semilla.
- Los archivos del wallet (claves cifradas con tu contraseña y caché de escaneo) son solo una comodidad. Se pueden regenerar con la semilla.

Gastar: por qué el saldo se "bloquea"

Cada salida nueva, incluido el cambio, tarda 10 bloques (unos 20 minutos) en poder gastarse. Si tu wallet tiene una sola salida grande y haces un pago, el cambio queda bloqueado y no puedes enviar otro pago hasta que se desbloquee. Tener varias salidas de tamaño razonable evita ese

cuello de botella.

Al construir una transacción el wallet elige qué salidas gastar, pide al nodo los señuelos (cap. 7), calcula la comisión (cap. 10), firma y la envía. Gastar muchas salidas pequeñas a la vez hace la transacción más grande y cara.

Variantes de wallet

TIPO	QUIÉN GUARDA LAS CLAVES	COMPROMISO DE PRIVACIDAD
Completo (GUI o CLI oficial, Feather)	Tu equipo	Máximo, si usas tu propio nodo
De hardware (Ledger, Trezor)	El dispositivo guarda la clave de gasto; la de vista se comparte con el equipo	Muy buena; se protege el gasto
Móvil con nodo remoto (Cake Wallet y similares)	Tu teléfono	Depende del nodo al que te conectes (cap. 12)
Servidor de wallet ligero	Entregas tu clave de vista a un tercero	El servidor ve todo lo que recibes y gastas
Solo lectura	Solo la clave de vista	Sirve para contabilidad y auditoría sin riesgo de gasto

Multifirma

Monero soporta firmas M de N (por ejemplo 2 de 3), y para construirlas los participantes intercambian varias rondas de datos antes de poder firmar. La función se considera experimental y hay que activarla de forma explícita en el wallet de consola. No la uses con fondos importantes sin entender sus límites.

CAPÍTULO 14

Ataques y límites de la privacidad

La privacidad de Monero rara vez se rompe atacando la criptografía; se erosiona con lo que la rodea: anillos pequeños, contrapartes que saben demasiado, metadatos de red y errores del usuario. Conocer estos límites es parte de entender el protocolo.

Catálogo de ataques

ATAQUE	QUÉ EXPLOTA	MITIGACIÓN
Heurística temporal ("el más nuevo es el real")	Señuelos elegidos sin imitar el patrón de gasto real	Distribución gamma (cap. 7); FCMP++ lo elimina (cap. 15)
Señuelos descartados por efecto en cadena	Salidas ya conocidas como gastadas dejan de contar como señuelo	RingCT y anillo mínimo obligatorios; anillo de 16
Inundación de salidas propias	Un atacante crea muchas salidas y sabe cuáles de un anillo son suyas, y por tanto no reales	Más tamaño de anillo; FCMP++
Eve-Alice-Eve (EAE)	La contraparte conoce el origen y el destino de tus fondos y puede acotar cuál entrada gastaste	Esperar entre recibir y gastar; no repetir patrones; FCMP++
Correlación con exchanges	Retiros y depósitos con identidad (KYC) enlazados por momento y cantidad aproximada	Separar plazos y cantidades; usar subdirecciones
Análisis de red	La IP desde la que sale tu transacción	Dandelion++, Tor o I2P, nodo propio (cap. 12)
Nodo remoto malicioso	Te sirve señuelos sesgados o mide tu actividad	Nodo propio; revisar distribución de señuelos
Huellas del wallet	Versiones o ajustes que producen transacciones distinguibles	Reglas de consenso uniformes; mantener el wallet actualizado

Cuánto reduce el anillo un atacante con salidas propias

Si cada señuelo tiene probabilidad `p` de ser una salida controlada por el atacante (y, por tanto, descartable), el tamaño efectivo del anillo es aproximadamente:

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

En palabras: cada señuelo que el atacante controla desaparece del anillo, así que quedan tantos miembros plausibles como señuelos no controlados, más el real.

Con `n = 16` y `p = 0,5`, quedan unos 8,5 miembros plausibles. La privacidad cae sin que el atacante rompa nada: solo reduce el conjunto en el que te escondes. Por eso aumentar `n`, y en última instancia usar toda la cadena, es una mejora real.

Errores que cometen los usuarios

- Publicar una dirección junto a tu identidad y recibir en ella, o mostrar capturas de comprobantes de pago.
- Compartir la clave de vista con terceros sin pensar que verán todos tus ingresos para siempre.
- Reutilizar la misma dirección con muchos contactos en vez de subdirecciones.
- Gastar siempre de forma inmediata y con el mismo patrón, lo que facilita la correlación temporal.
- Mezclar Monero con servicios que exigen identidad sin tener en cuenta que la otra parte conserva lo que ve.

Qué no promete Monero

La cadena oculta quién paga a quién y cuánto, pero no protege lo que tú revelas fuera de ella. No oculta que una transacción existió, ni su tamaño, ni el momento. Y un adversario con acceso a tu equipo, a tu teléfono o a tu conexión de red tiene más información que cualquiera que solo lea la cadena.

CAPÍTULO 15

El futuro: FCMP++ y Carrot

FCMP++ sustituye el anillo de 16 miembros por una prueba de que tu salida está en el conjunto de todas las salidas de la cadena, y Carrot renueva el esquema de claves y direcciones. A día de hoy, noviembre de 2026, siguen en fase de pruebas y no tienen fecha de activación confirmada en mainnet.

Estado actual

Según Bitrue (12 de octubre de 2026), el hard fork del 5 de octubre ocurrió en la red de pruebas *stressnet* en el bloque 3.102.800 y no afectó a la mainnet ni a los saldos. La misma fuente indica que no hay fecha confirmada para el hard fork principal y que una hoja de ruta no vinculante apunta a la semana 16 de 2027. Alguna página de baja calidad afirma que FCMP++ ya está activo; las fuentes recientes y detalladas dicen lo contrario. Antes de depender de una fecha, comprueba el anuncio oficial del proyecto.

La idea de FCMP++

Con anillos, el anonimato es el de 16 candidatos. Con FCMP++ el firmante demuestra, sin revelar cuál, que su salida pertenece al conjunto completo de salidas existentes, que la fuente citada cifra en más de 150 millones. Ya no se eligen señuelos, así que desaparece la superficie de ataque del capítulo 7.

Para que sea viable, el diseño combina tres piezas:

- **Árbol de curvas (*curve tree*).** Todas las salidas se organizan en un árbol tipo Merkle cuyos niveles alternan entre dos curvas elípticas emparejadas, de modo que las pruebas de pertenencia son pequeñas y no dependen de lo grande que sea la cadena.
- **Prueba de pertenencia en conocimiento cero.** Demuestra que un camino del árbol llega a una salida existente, sin decir cuál. Utiliza una variante de Bulletproofs.
- **Prueba de autorización y enlace.** Una versión re-aleatorizada de la salida demuestra que conoces su clave y que la key image es la correcta. La key image se mantiene: sigue haciendo falta para detectar el doble gasto.

Qué cambia y qué no

ASPECTO	HOY (CLSAG, ANILLO DE 16)	CON FCMP++
Conjunto de anonimato	16 salidas	Todas las salidas de la cadena
Selección de señuelos	Necesaria y vulnerable a ataques estadísticos	No existe
Key images y doble gasto	Sí	Sin cambios
Montos ocultos (Pedersen y Bulletproofs+)	Sí	Sin cambios
Tamaño de la prueba por entrada	576 bytes	Mayor, y crece poco con el tamaño de la cadena
Nodos	Guardan salidas y key images	Mantienen además el árbol de curvas
Wallets	Piden señuelos al nodo	Piden la rama del árbol de su salida; con un nodo remoto conviene saber qué revela

No encontré cifras fiables de tamaño y coste de verificación para citarlas, así que no las incluyo.

Carrot

Carrot es el nuevo esquema de direcciones y claves pensado para acompañar a FCMP++. Su especificación separa capacidades que hoy están mezcladas, como generar direcciones, ver lo recibido, ver el saldo completo y autorizar gastos, para poder dar a cada herramienta solo la

clave que necesita. También refuerza el protocolo frente a ataques en los que alguien engaña a un receptor para identificar su dirección.

Para los usuarios habrá que actualizar el wallet antes del hard fork. Los detalles de migración (formato de dirección, semillas existentes) deben confirmarse en la documentación oficial cuando se aproxime la fecha.

Seraphis y Jamtis

Seraphis y Jamtis fueron una propuesta anterior y más ambiciosa para rediseñar transacciones y direcciones. FCMP++ y Carrot recogen parte de esas ideas con un camino de implementación más corto. Que Seraphis llegue a desplegarse sigue siendo incierto.

CAPÍTULO 16

Laboratorio: ver el protocolo funcionando

Cuatro ejercicios opcionales, de menor a mayor dificultad, para comprobar con tus propios ojos lo que has leído. Los dos primeros no requieren programar; el tercero incluye un script completo y probado.

Ejercicio 1: levantar un nodo en stagenet

La *stagenet* es una red de pruebas con las mismas reglas que la mainnet y monedas sin valor, ideal para experimentar sin riesgo. Descarga los binarios oficiales de Monero, verifica sus firmas y ejecuta:

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

Cuando haya sincronizado (o ya tenga bloques), pregunta al nodo por su estado vía RPC. En stagenet el puerto RPC es el 38081 (en mainnet, el 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"}'
```

Busca en la respuesta la altura, la dificultad y la versión del hard fork activo.

Ejercicio 2: disecar una transacción real

Elige un identificador de transacción de un explorador de bloques de stagenet y pídelo al nodo:

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

En el JSON, los nombres exactos varían un poco entre versiones, pero deberías poder comprobar:

- cuántos miembros tiene el anillo de cada entrada (`key_offsets`) y dónde está la key image (`k_image`);
- que las salidas tienen `amount` igual a 0 y llevan clave de un solo uso y *view tag*;
- que el tipo RingCT es 6 y que hay un compromiso (`outPk`) por salida y un `pseudoOut` por entrada;
- que hay al menos dos salidas, y cuánto pesa cada parte frente a la tabla del capítulo 10.

Ejercicio 3: derivar y escanear una salida sigilosa

Este script implementa desde cero, con aritmética de curva en Python puro, lo descrito en los capítulos 3, 4 y 8. Comprueba que emisor y receptor llegan al mismo secreto compartido, que el *view tag* coincide, que la clave privada de la salida recupera la clave pública, y que los compromisos de Pedersen se suman. Necesita `pip install pycryptodome` (para Keccak-256).

```python
# stealth_demo.py: direcciones sigilosas y compromisos de Pedersen sobre Ed25519.
# No es una wallet: no codifica direcciones ni firma transacciones.
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)  # raíz cuadrada de -1

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

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

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

# --- Aritmética de la curva (coordenadas extendidas, 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):                          # multiplicación escalar
    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. Claves del receptor ---
b = rand_scalar()                                   # clave privada de gasto
a = Hs(b.to_bytes(32, "little"))                    # clave privada de vista
B, A = mul(b, G), mul(a, G)

# --- 2. El emisor crea la salida k ---
k = 0
r = rand_scalar()
R = mul(r, G)                                       # clave pública de la tx
D_sender = mul(8, mul(r, A))                        # derivación = 8·r·A
P = add(mul(Hs(encode(D_sender), varint(k)), G), B) # clave de un solo uso
tag = keccak256(b"view_tag" + encode(D_sender) + varint(k))[0]

# --- 3. El receptor escanea ---
D_recv = mul(8, mul(a, R))                          # derivación = 8·a·R
assert encode(D_recv) == encode(D_sender), "Diffie-Hellman no coincide"
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         # clave privada de la salida
spendable = encode(mul(x, G)) == encode(P)

print("view tag coincide :", tag_ok)
print("la salida es mía  :", mine)
print("x*G == P          :", spendable)

# --- 4. Pedersen y su suma (H de juguete: NO seguro, solo para ver la linealidad) ---
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(suma):", encode(lhs) == encode(rhs))
```

El resultado esperado son cuatro líneas con `True`. Dos advertencias: la `H` de juguete (`7·G`) hace que cualquiera conozca su logaritmo discreto respecto a `G`, así que ese compromiso no es vinculante; y Monero deriva la `H` real por hash a punto, algo que este script no implementa.

Ejercicio 4 (avanzado): qué se rompe si cambias algo

Modifica el script y razona qué observas antes de ejecutarlo:

1. Cambia el índice `k` del receptor por `1`. ¿Sigue siendo suya la salida?
2. Escanea con la clave de vista de otra persona. ¿Qué ocurre con el *view tag*? ¿Y con `P_check`?
3. Añade una segunda salida con otro `k` y comprueba que sus claves de un solo uso no guardan relación visible.
4. Reto: implementa `Hp` (hash a punto) y calcula la key image `x·Hp(P)`. Valida tu implementación contra los vectores de prueba del repositorio oficial de Monero.

CAPÍTULO 17

Referencias y lecturas recomendadas

Estado del protocolo a noviembre de 2026. Las cifras de red cambian: contrasta siempre con las fuentes oficiales del proyecto antes de tomar decisiones.

Fuentes del estado actual (capítulos 11 y 15)

Estas páginas se consultaron mediante búsqueda web, y de ellas sale lo relativo a FCMP++ y a las reorganizaciones de 2025:

- When Is the Next Monero Hard Fork? (Bitrue, 12 de octubre de 2026): estado de FCMP++ y fecha de la stressnet.

- Monero faces chain reorganization fears after Qubic says it controls 51% of hashrate (The Block): reorganización de seis bloques, agosto de 2025.
- Monero rallies 7% hours after 18-block reorg (Cointelegraph): reorganización de 18 bloques, septiembre de 2025.

Lecturas recomendadas

No se abrieron durante la redacción; son las referencias estándar del tema y conviene verificar que los enlaces siguen vigentes.

TEMA	LECTURA
Visión completa del protocolo	*Zero to Monero* (Alonso, Noether, Goodell), en getmonero.org
Origen del diseño	*CryptoNote v2.0* (van Saberhagen, 2013)
Firmas de anillo CLSAG	Goodell, Noether y RandomRun, *Concise Linkable Ring Signatures and Forgery Against Adversarial Keys*, IACR ePrint 2019/654
Pruebas de rango	*Bulletproofs* (Bünz et al., ePrint 2017/1066) y *Bulletproofs+* (Chung et al., ePrint 2020/735)
Análisis de trazabilidad	Möser et al., *An Empirical Analysis of Traceability in the Monero Blockchain*, arXiv 1704.04299
Red	Fanti et al., *Dandelion++*, arXiv 1805.11060
PoW	Repositorio `tevador/RandomX` en GitHub, con su especificación
Árboles de curvas (base de FCMP++)	Campanelli, Hall-Andersen y Kamp, *Curve Trees*, ePrint 2022/756
Código de referencia	Repositorio `monero-project/monero` en GitHub
Investigación abierta	Monero Research Lab, en getmonero.org

Cómo seguir aprendiendo

Lee primero *Zero to Monero* con esta guía al lado, para contrastar cada concepto con una explicación más extensa. Después, el ejercicio 4 del laboratorio es la mejor forma de comprobar que entiendes la key image y la firma de anillo.