# A Tour of the Monero Project

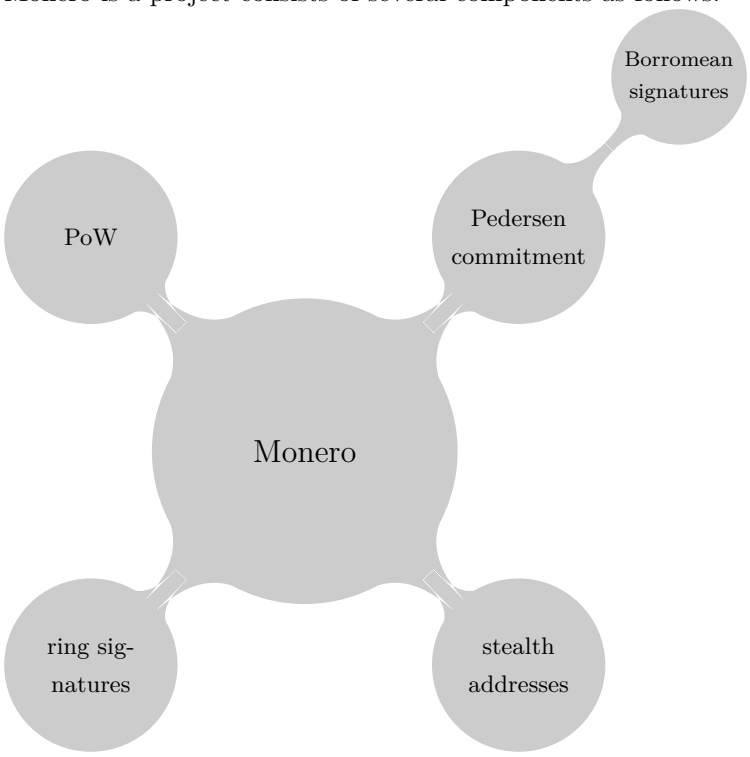
Sammy

September 22, 2017

# Contents

# Chapter 1. Overview

Monero is a project consists of several components as follows.



# Chapter 2. Monero Project

## 2.1 RPC Documentation

### 2.1.1 Wallet

The RPC interfaces are documented according to their namespaces as follows.

**wallet_rpc_server**

Table 2.1: The JSON RPC and their corresponding callbacks for wallet

| JSON RPC | Callback |
|---|---|
| `make_integrated_address` | `on_make_integrated_address` |

## 2.2 API

Every column of the key matrix will consist of keys from a single person in this part.

**proveRctMG**

**Input**

- $msg$: message to sign
- $\mathbf{inPk} = \left[(pk_{j,i}, c_{j,i})\right]_{m \times n}$: input public key matrix of $m$ rows and $n$ columns
- $\mathbf{inSk} = \big((sk_1, x_1^c), \ldots, (sk_j, x_j^c), \ldots, (sk_m, x_m^c)\big)$: input secret key vector of size $m$
- $\mathbf{outSk} = (y_1, \ldots, y_k, \ldots, y_{m'})$: output secret key vector of size $m'$
- $\mathbf{outPk} = \Big((\hat{pk}_1, \hat{c}_1), \ldots, (\hat{pk}_k, \hat{c}_k), \ldots, (\hat{pk}_{m'}, \hat{c}_{m'})\Big)$: output public key vector of size $m'$
- $\pi$: the column index of public key matrix corresponds to the secret key vector **inSk**
- $bH$: i.e. key for transaction fee of amount $b$ is the paper

**Ouput** the multi-layer ring signature of the input message $msg$

**Procedure** 1. for $i = 1, \ldots, (m+1)$, $sk_i^* = 0$, where $sk^*$ is a vector of key

2. initialize a key matrix as

$$\mathbf{M} = \begin{bmatrix} I_1 & \dots & I_1 \\ \vdots & \vdots & \vdots \\ I_j & \dots & I_j \\ \vdots & \vdots & \vdots \\ I_{m+1} & \dots & I_{m+1} \end{bmatrix}_{(m+1)\times n}$$

where vector $I = (1, 0, 0, \dots, 0)$ of length 32 is a key corresponding to the zero elliptic curve point

3. update $\mathbf{M}$ as

$$\mathbf{M} = \begin{bmatrix} pk_{1,1} & \dots & pk_{1,i} & \dots & pk_{1,n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ pk_{j,1} & \dots & pk_{j,i} & \dots & pk_{j,n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ pk_{m,1} & \dots & pk_{m,i} & \dots & pk_{m,n} \\ \sum_{j=1}^{m} c_{j,1} & \dots & \sum_{j=1}^{m} c_{j,i} & \dots & \sum_{j=1}^{m} c_{j,n} \end{bmatrix}_{(m+1)\times n}$$

4. update $sk^* = (sk_1, \dots, sk_j, \dots, sk_m, \sum_{j=1}^{m} sk_j)$

5. update $\mathbf{M}$ as

$$\mathbf{M} = \begin{bmatrix} pk_{1,1} & \dots & pk_{1,i} & \dots & pk_{1,n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ pk_{j,1} & \dots & pk_{j,i} & \dots & pk_{j,n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ pk_{m,1} & \dots & pk_{m,i} & \dots & pk_{m,n} \\ \dots & \dots & \sum_{j=1}^{m} c_{j,1} - \sum_{k=1}^{m'} \hat{c}_k - bH & \dots & \sum_{j=1}^{m} c_{j,n} - \sum_{k=1}^{m'} \hat{c}_k - bH \end{bmatrix}_{(m+1)\times n}$$

6. update $sk^* = (sk_1, \dots, sk_j, \dots, sk_m, \sum_{j=1}^{m} sk_j - \sum_{k=1}^{m'} y_k)$

7. hand over $msg$, $\mathbf{M}$, $sk^*$, $\pi$ and $m$ to function `MLSAG_Gen`, (i.e., treat all public keys but the last as double-spendable) and return the signature by `MLSAG_Gen`

**proveRctMGSimple**

**Input**

- $msg$: message to sign
- $inPK = ((pk_1, c_1), (pk_2, c_2), \dots, (pk_m, c_m))$: input public key vector of length $n$, where $pk_i$ is the public key, and $c_i$ is the commitment
- $inSK = (sk, x^c)$: input secret key, where $sk$ is the actual secret key, and $x^c$ is the mask key
- $y$: output secret key, i.e. the mask value
- $\hat{c}$: output public key, i.e., the output commitment
- $\pi$: index of the public key in $inPK$ paired with $inSK$

**Ouput** the simple(one-layer) ring signature of the input message $msg$

**Procedure**

1. initialize $\mathbf{M} = []_{2\times n}$, a $2 \times n$ matrix

2. update

$$\mathbf{M} = \begin{bmatrix} \dots & pk_i & \dots \\ \dots & c_i & \dots \end{bmatrix}$$

and then

$$\mathbf{M} = \begin{bmatrix} \dots & pk_i & \dots \\ \dots & c_i - \hat{c} & \dots \end{bmatrix}$$

3. estimate $sk^* = (sk, sk - y)$
4. hand over $msg$, $\mathbf{M}$, $sk^*$, $\pi$ and 1 to function `MLSAG_Gen`, (i.e., treat all public keys but the last as double-spendable) and return the signature by `MLSAG_Gen`

**MLSAG_Gen**

**Input**
- $msg$: message to sign
- $\mathbf{Y} = \left[y_{j,i}\right]_{m \times n}$: the public key matrix of $m$ rows and $n$ columns
- $\mathbf{x} = (x_1, x_2, \dots, x_m)$: the secret key vector of length $m$ from the singer
- $\pi$: the column index of public key matrix corresponds to the secret key vector $\mathbf{x}$
- $m'$: the number of double spendable rows, should satisfy $m' < m$

**Output** signature as $(I_1, \dots, I_m, c_1, \mathbf{S} = [s_{j,i}]_{m \times n})$ where $I_j$ is the key image, $c_1$ is the first hash (treated as a scalar) in the ring, and $\mathbf{S}$ is the scalar matrix of randomly generated to make up the ring. (c.f.g struct `mgSig` in the `rctTypes.h`)

**Procedure**
1. for $j \leftarrow 1, \dots, m'$, generate $\alpha_j$ randomly and estimate
   - $L_j \leftarrow \alpha_i G$
   - $H_{P_j} \leftarrow hashToPoint(Y_{j,\pi})$
   - $R_j \leftarrow \alpha_j H_{P_j}$
   - $I_j \leftarrow x_j H_{P_j}$
2. for $j \leftarrow (m'+1), \dots, m$, generate $\alpha_j$ randomly and estimate $L_j \leftarrow \alpha_i G$
3. calculate $c_{old} \leftarrow H(msg, Y_{1,\pi}, L_j, R_j, \dots, Y_{m',\pi}, L_{m'}, R_{m'}, Y_{(m'+1),\pi}, L_{(m'+1)}, \dots, Y_{m,\pi}, L_m)$
4. initialize $i \leftarrow (\pi + 1) \bmod n$, if $i = 0$, set $c_1 \leftarrow c_{old}$
5. for $i \leftarrow (\pi+1), \dots, n, 1, \dots, (\pi - 1)$
   (a) generate $s_{j,i}$ randomly for $j \leftarrow 1, \dots, m$
   (b) for $j \leftarrow 1, \dots, m'$, calculate
      - $L_j \leftarrow s_{j,i} G + c_{old} Y_{j,i}$
      - $R_j \leftarrow s_{j,i} hashToPoint(Y_{j,i}) + c_{old} I_j$
   (c) for $j \leftarrow (m'+1), \dots, m$, estimate $L_j \leftarrow s_{j,i} G + c_{old} Y_{j,i}$
   (d) update $c \leftarrow H(msg, Y_{1,i}, L_j, R_j, \dots, Y_{m',i}, L_{m'}, R_{m'}, \dots, Y_{(m'+1),i}, L_{(m'+1)}, \dots, Y_{m,i}, L_m)$
   (e) increment $i \leftarrow (i+1) \bmod n$, if $i = 0$, set $c_1 \leftarrow c_{old}$
   (f) update $c_{old} \leftarrow c$
6. update $s_{j,\pi} \leftarrow \alpha_j - c x_j$

**construct_tx_and_get_tx_key**

**Input**

- **ack**: account key of the payer, consisting of two seckey-pubkey pairs as $(a, A = a \cdot G)$ for viewing and $(b, B = b \cdot G)$ for spending
- **inCoin** $= \{inCoin_i = (amt_i, \{(P_{i,j}, C_{i,j}, o_{i,j})\}_{j=1}^{l_i}, R_i, k_i, k'_i)\}_{i=1}^{m}$: input coin ring vector, where for each $inCoin_i$
  - $amt_i$ is the amount of the real input coin
  - $P_{i,j}$ is the one-time address of the j-th coin
  - $C_{i,j}$ is the commitment of amount of the j-th coin
  - $o_{i,j}$ is the global output index of the j-th coin
  - $R_i$ is the key for the tx producing the real input coin
  - $k_i$ is the index of the real input coin
  - $k'_i$ is the index of the real input coin in the tx containing it
- **outDest** $= \{(\hat{amt}_j, A_j, B_j)\}$: output destination vector, where for the j-th destination, $\hat{amt}_j$ is the amount to send, $A_j$ is the public key for viewing, and $B_j$ is the public key for spending
- *extra*: extra field storing payment ID, public tx key, etc

**Output**

- **tx**: the constructed tx with RCT signature set up

**Procedure**

1. generate a tx key pair $(r, R = r \cdot G)$
2. remove pubkeys in *extra* if any and add $R$ to *extra*
3. encrypt the stealth payment ID if any in *extra* with $r \cdot A$
4. For each $inCoin_i$,
   - derive the address key pair $(x_i^*, P_i^*) = (H_s(a \cdot R_i || k'_i) + b, H_s(a \cdot R_i || k'_i) \cdot G + B)$ and key image $I_i = x_i^* \cdot H_p(P_i^*)$
   - ensure if $P_i^*$ is equal to the one $P_{i,k_1}$ specified in $inCoin_i$
   - make an input entry as $inToKey_i = (amt_i^*, I_i, \{o_{i,1}^* = o_{i,1}, o_{i,2}^* = o_{i,2} - o_{i,1}, \ldots, o_{i,j}^* = o_{i,j} - o_{i,j-1}, \ldots\})$
5. sort $outDest_i \in$ **outDest** in ascending order of their amount
6. For each $outDest_j$,
   - compute the amount key $amtKey_j = H_s(r \cdot A_j || j)$
   - calculate the one-time key $\hat{P}_j = amtKey_j \cdot G + B_j$
   - make an output coin as $outCoin_j = (\hat{amt}_j, \hat{P}_j)$
7. assert $\sum amt_i > \sum \hat{amt}_j$
8. If the number $m > 1$ of input coin ring, use simple RCT, or else use the full one
   - For simple RCT,
     (a) $\mathbf{M} = []_{1 \times m}$
     (b) compute the real input amount key and commitment $\{(x_i, C_i)\}_{i=1}^{m}$
     (c) update the i-th column of $\mathbf{M}$ as $M_i = [(P_{i,j}, C_{i,j})]_{j=1}^{l_i}$
     (d) save a copy $\{amt_i^*\}$ of input amount $\{amt_i\}$
     (e) save a copy $\{\hat{amt}_j^*\}$ of output amount $\{\hat{amt}_j\}$

(f) mask the amount in inputs $inToKey_i$ and outputs $outCoin_j$ by zeroing them

(g) compute the hash $h_{pre}$ of the tx prefix including the version, unlock time, $\{inToKey_i\}$, $\{outCoin_j\}$ and extra nonces

(h) delegate the signing job to `rct::genRctSimple` with message $h_{pre}$, input secret key vector $\{(x_i, C_i)\}$, destination $\{outCoin_j\}$, input amount $\{amt_i^*\}$, output amount vector $\{\hat{amt}_j^*\}$, tx fee $(\sum amt_i^* - \sum \hat{amt}_j^*)$, mix-in matrix $\mathbf{M}$, amount key vector $\{amtKey_i\}$, real input index $\{k_i\}$ and get back mask values and MLSAG

- For non-simple RCT,

(a) $\mathbf{M} = []_{1 \times l_1}$

(b) $\mathbf{M} = [(P_{1,i}, C_{1,i})]_{1 \times l_1}$

(c) save a copy $\{amt_i^*\}$ of input amount $\{amt_i\}$

(d) save a copy $\{\hat{amt}_j^*\}$ of output amount $\{\hat{amt}_j\}$

(e) append tx fee $(amt_1^* - \sum \hat{amt}_j^*)$ to $\{\hat{amt}_j^*\}$

(f) mask the amount in inputs $inToKey_i$ and outputs $outCoin_j$ by zeroing them

(g) compute the hash $h_{pre}$ of the tx prefix including the version, unlock time, $\{inToKey_i\}$, $\{outCoin_j\}$ and extra nonces

(h) delegate the signing job to `rct::genRct` with message $h_{pre}$, input secret key vector $(x_1, C_1)$, destination $\{outCoin_j\}$, output amount vector $\{\hat{amt}_j^*\}$, mix-in matrix $\mathbf{M}$, amount key vector $\{amtKey_i\}$, real input index $k_1$ and get back mask values and MLSAG

**Input** •

**Output**

**Procedure** 1.

# Chapter 3. Blog Series about XMR

## 3.1 Interesting Sites

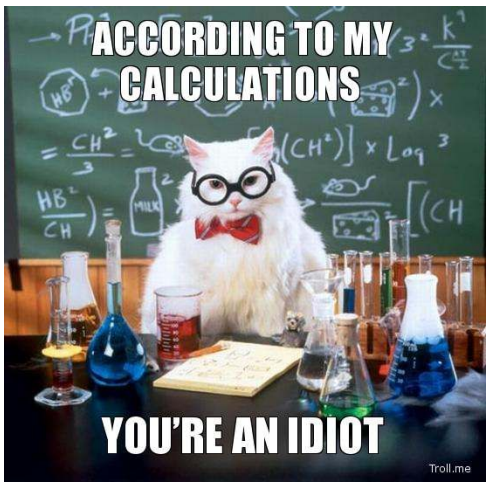
- https://moneroeric.com/monero-sites/
- https://monero.stackexchange.com/

## 3.2 Understanding Monero Cryptography, Privacy – Introduction

**Cited from** XMR crypto blog series by luigi1111.

> This is part two of a series of unknown size; it'll be done when it's done. Part one is here.
>
> Part one focuses on the basics: ECC, the particular curve, private and public "keys", and a bonus section on how Monero addresses are generated.
>
> Note: Monero is based on the Cryptonote protocol – though it has diverged and will continue to diverge – along with numerous other coins; much of this series applies equally well to the others with some caveats. Monero is easily the largest and most active Cryptonote-based project.

Hello! I'm an autodidact enthusiast of cryptography, particularly in relation to the crypto-currency Monero. Naturally, you should not assume everything I say is correct, and I hope any egregious errors are pointed out so I can fix them (and help my own understanding). Just calling me an idiot is fine too.



Monero's tagline is "Secure, Private, Untraceable." Secure could refer to a number of facets of a crypto-currency, but here we are only particularly interested in security relating to privacy/anonymity. These articles will be looking at how Monero achieves "privacy", that is unlinkability and untraceability, with references to security where appropriate. This article focuses on some concepts, which will hopefully make understanding the others easier. Without further ado, let's get into it!

### 3.2.1 What is Elliptic Curve Cryptography?

Alright, so what is ECC? From Wikipedia: "**Elliptic curve cryptography (ECC)** is an approach to public-key cryptography based on the algebraic structure of elliptic curves over finite fields."

Now, what does that mean? *I have no idea.*

More seriously, let's go through it:

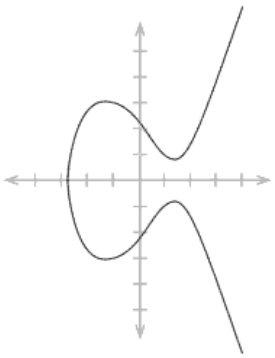
1. Public-Key Cryptography, or asymmetric cryptography, uses a pair of keys instead of a single private key as in symmetric cryptography (e.g., AES): a public key, to be given out to "the world"; and a private key, to be always kept secret. To be secure, it must be hard intractable to figure out the private key given the public key; to be usable it must be easy to calculate the public key given the private key. ECC relies on the ECDLP for its security. **Takeaways: public/private key pair; private->public is easy, but public->private is "impossible".**

2. "algebraic structure of elliptic curves": *What is this???* It is a plane curve satisfying $y^2 = x^3+ax+b$. It might look something like Figure 3.1a

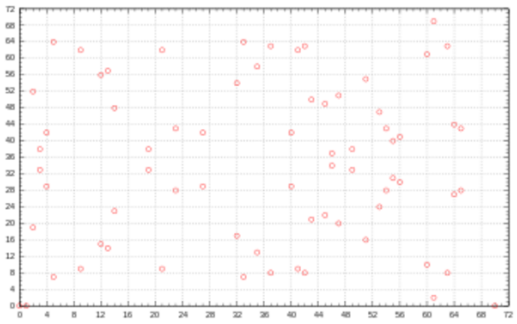
   *Who cares?* Right, probably no one. In case someone does, there's a wealth of articles (many related to Bitcoin) out there that explain in detail how they work, how addition is possible, etc. Some examples: A, B, C (a series itself). Numerous videos are out there too, if you're into that. **Takeaways: none, this funny-looking curve will not help you understand and isn't even how Monero's curve looks.**

3. "over finite fields": this just means curve points are taken modulo some (large, prime) number. Everyone is familiar with modular addition and subtraction at least (even if they've never heard the word) due to our time-keeping. *If it is 10am, what time will it be in 5 hours?* **Congrats, you just did modular addition.** An elliptic curve over a finite field might look something like Figure 3.1b

   *Whoa, that looks odd.* Yes, it does. **Takeaways: none. Actually, note how the points are "reflected" over an invisible line in the center.**



(a) An example of elliptic curves



(b) An example of elliptic curves

A primary benefit of using ECC vs something like RSA is that keys are much smaller for similar security levels.

**I believe the only things you need to know to proceed are:**

1. A point on the curve can be added to or subtracted from another point, or itself.
2. A point cannot be multiplied or divided by another point.
3. Adding a point to itself allows "scalar multiplication", **which is where the magic happens**.

Subtracting a point from itself isn't very useful, as it'll just return the ECC equivalent of 0. Division by integer isn't possible (the equivalent modular operation – modular multiplicative inverse – is, but only with knowledge of the original scalar).

Scalar multiplication is just adding a point to itself over and over; given a point $A$, $5A = A + A + A + A + A$. Since we use astronomically large scalars to prevent easy brute-forcing, we use techniques like double-and-add to allow computation in near-logarithmic time (i.e., really fast!). A quick example:

Suppose our scalar is 27, and we want to compute $27A$. Using the naive method, we'd need 26 additions. Instead:

1. Add $A$ to itself: $2A$. Let's call this new point B.
2. Add $B$ to itself: $2B = 4A = C$.
3. Add $C$ to itself: $2C = 4B = 8A = D$.
4. Add $D$ to itself: $2D = 4C = 8B = 16A = E$.
5. Add $D$ to $E$: $24A = F$.
6. Add $B$ to $F$: $26A = G$.
7. Add $A$ to $G$ : $27A$

We went from 26 additions to 7. The difference grows exponentially with larger scalars. The speed difference for an average-size scalar is something along the lines of "all the energy in the universe isn't enough" and "takes less than 1/100th of a second on an average computer", which is interesting to ponder.

That's it for general ECC stuff! If you want more in-depth technical details, please see the links above. :)

### 3.2.2 The Monero Curve and Private and Public "Keys"

Now, onto the Monero-specific stuff. Finally.

First some boring stuff like curve constants. From the Cryptonote whitepaper, we get:

- $q$: a prime number; $q = 2^{255} - 19$
- $d$: an element of $\mathbb{F}_q$; $d = -121665/121666$
- $E$: an elliptic curve equation; $-x^2 + y^2 = 1 + dx^2y^2$
- $G$: a base point; $G = (x, -4/5)$

- $l$: a prime order of the base point; $l = 2^{252} + 27742317777372353535851937790883648493$;
- $H_s$: a cryptographic hash function $0, 1^* \to \mathbb{F}_q$
- $H_p$: a deterministic hash function $E(\mathbb{F}) \to E(\mathbb{F}_q)$

We are dealing with the Ed25519 curve, which is a Twisted Edwards Curve. *Good, more meaningless details!*

**Let's quickly go through it:**

- $q$: this is the total number of points on this curve. It is mostly irrelevant for our purposes.
- $d$: an element used in the curve equation below. Not important.
- $E$: the equation for our Ed25519 curve. *Wow, shiny!* Not important.
- $G$: the base point or generator point. **This is important!** It is the base from which many operations start. It is the "A" in the above example. In hex, which all of our keys are commonly represented in, it looks like: "5866666666666666666666666666666666666666666666666666666666666666". Great, back to useless information.
- $l$: the "order" of the above base point. **This is important**, as it defines the maximum number of points we can use, and the maximum size our scalars can be. This number is like the number "12" to a clock; adding points or scalars together that would "go over" means they will "wrap around" instead. If you could add $G$ to itself over and over and over until you reached l-1 number of additions, you would end up back at $G$.
- $H_s$ and $H_p$: s means scalar, $p$ means point. These will be discussed in a later article.

**Note:**

1. Scalars (private keys, really just large integers) are always represented by lowercase letters in equations.
2. Points (public keys, really an encoded coordinate on the curve) are always represented by uppercase letters.

In the "real world" (user-facing), both private and public keys in Monero are represented by 64 hex characters, similar to the above representation of $G$. Time for more useless information. Scalars are straightforwardly represented as little-endian integers (any integer between 0 and $l$ is valid), while points are specially encoded in a way that is too complex for this article. *Or maybe I haven't cared enough about the encoding to research it.*

If we use $x$ as our private key and P as our public key, then $P = xG$.

Some "fun" examples:

1. $x = 1$ or "0100000000000000000000000000000000000000000000000000000000000000" (remember little-endian); $P$ = "5866666666666666666666666666666666666666666666666666666666666666" or $G$. ($1G = G$)
2. $x = l - 1$ or "ecd3f55c1a631258d69cf7a2def9de1400000000000000000000000000000010"; $P$ = "5866666666666666666666666666666666666666666666666666666666666666e6" (note similarity to $G$); This is the last point before wrapping around. You can think of it like $-G$. Adding $G$ to this value will produce a special identity element, the same as multiplying a point by 0 or order $l$, or subtracting a point from itself.

3. The integer $(l + 1)/2$, "f7e97a2e8d31092c6bce7b51ef7c6f0a00000000000000000000000000000008", produces the point farthest away from $G$ (close enough, it and the next point are tied due to $l$ being odd), "ac1999070321b2c6309cc8e31aa89a8b3baa75b5f8febf47855555a3e744bcf0", similar to how 6 is farthest away from 12 on a clock. It (just like $G$ and **every** other point) has a complimentary point (in this case produced by $(l - 1)/2$), with which it will sum to the identity element.

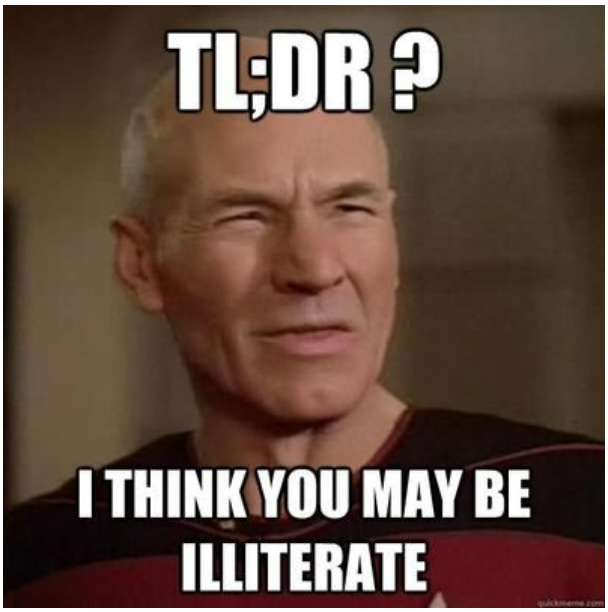
### 3.2.3 Monero Accounts and Addresses

This has gone a little long, so I'll just briefly restate the information available here for the standard deterministic derivation. The reason we have two key pairs will be discussed in a future article on stealth addresses.

1. Choose a random private spend key, typically by creating 256 random bits then "reducing" mod $l$. Call this key $b$ (to match the whitepaper – it's confusing I know).
2. Hash $b$ with the chosen algorithm, $H$ (Keccak_256 in our usage). Interpret the result as an integer and reduce it mod l as before. Call this key $a$.
3. Calculate $B = bG$ and $A = aG$. These are your public spend and public view keys.
4. Hash (prefix (0x12 in standard Monero) + $B$ + $A$) with $H$.
5. Append the first four bytes of the result to (prefix + $B$ + $A$). This will be 69 bytes (1 + 32 + 32 + 4).
6. Convert to cnBase58. This is not as straightforward as regular base58, as it uses blocks and padding to result in fixed-length conversions. 69 bytes will always be 95 cnBase58 characters.

Integrated addresses (described here) are the same as above, but with an 8 byte Payment ID appended to $A$ in step 4 above and a different prefix (0x13).

That does it for the introduction! Hopefully it wasn't completely incoherent rambling. Additional articles on stealth addresses and ring signatures will be coming out sometime soon. **Feedback is appreciated.**



For those of you looking for a TL;DR (or if you're just bored out of your mind), I've included a random picture (but no TL;DR).

Until next time!

## 3.3 Understanding Monero Cryptography, Privacy Part 2 – Stealth Addresses

**Cited from** blog by luigi1111.

> This is part two of a series of unknown size; it'll be done when it's done. Part one is here.
>
> Part two focuses on stealth addresses, an essential part of the protocol.
>
> Note: Monero is based on the Cryptonote protocol – though it has diverged and will continue to diverge – along with numerous other coins; much of this series applies equally well to the others with some caveats. Monero is easily the largest and most active Cryptonote-based project.

Hello! I am back with part two of my series on Monero cryptography and privacy; in this series, I'm attempting to make the concepts as easy to understand as possible. I want you to have the same "lightbulb" moments I had when I first "got" these concepts (my understanding has evolved over a period of time).

In part two, we will be discussing stealth addresses, though you may learn some new concepts along the way. Stealth addresses are one of the two complementary techniques used to provide sender/receiver privacy in Monero.

So, what are stealth addresses? Well, they look like this:

(just kidding)

In my own words, stealth addressing is a technique whereby a **sender** can take a **recipient's** public address and transform it to a one-time address such that:

1. it is **publicly unlinkable** to the original public address;
2. it is **publicly unlinkable** to **any** other one-time address;
3. only the **recipient** can link all their payments together
4. only the **recipient** can derive the secret key associated with the one-time address

Using stealth addressing, a recipient can publish one address and receive unlimited* publicly unlinkable payments.

*[The chance of a collision (two stealth addresses being the same) is cryptographically negligible. Using the Birthday Paradox we can roughly estimate it would take $\sqrt{l}$, or about $2^{126}$, stealth addresses being created before having a 50% chance of a collision. The result would be that the colliding addresses become publicly linkable to each other, but not to any others. There is another, worse problem that would occur in Monero if two stealth addresses were to collide; this will be explained in the ring signature article.

For an analogy on how large $2^{126}$ is, imagine the world has 10 billion people. Each and every person sends a payment to **every** other person once per **second**. We would reach $2^{126}$ payments in about 27 billion **years**.]

Stealth addresses can be implemented by any currency, including Bitcoin, but **by themselves** do not provide significant extra privacy over avoiding address reuse. However, they are very handy for certain uses, e.g., a published donation address, where address reuse is basically unavoidable.

Now you know what stealth addresses **are**, but you probably don't yet "get" them or how they work. Unless you understood previously, in which case why are you reading this article?

Before getting into the specifics of stealth addresses in Monero, we need to discuss something:

### 3.3.1 ECDH

Elliptic Curve Diffie-Hellman is a variant of the original Diffie-Hellman key agreement protocol extended for use with ECC. In simple terms, two parties can independently generate a shared secret

over an unsecured connection (implying that no observer can discover the secret by simply watching their communication). Basic ECDH is quite simple to understand with the application of the scalar multiplication technique from my previous article. For simplicity all key pairs will always be referred to as a lowercase/capital letter (private key $a$, corresponding public key $A$, etc).

First, we have Alice. Alice has chosen a random private key (scalar) $a$ from our group $[1, l-1]$. Her public key is $A = aG$.

Similarly, Bob chooses random private key $b$. His public key is $B = bG$.

Knowing we can add points together, Alice could compute point $C = A + B$, but so could any observer. $C$ is **shared**, but it isn't **secret**.

Instead, remembering that $A$ and $B$ are curve points, and that we can add a point to itself (scalar multiplication!), Alice computes point $D = aB$. This is just like computing $B = bG$, but with a different "base point". Knowing the result $D$ and the base $B$ is no different with respect to helping an observer learn a than knowing result $A$ and base $G$!

Bob, in turn, can also compute $D' = bA$ (using the apostrophe to be a second "try" at the same thing). Now Alice and Bob have a shared, secret point known only to them! $D = D'$

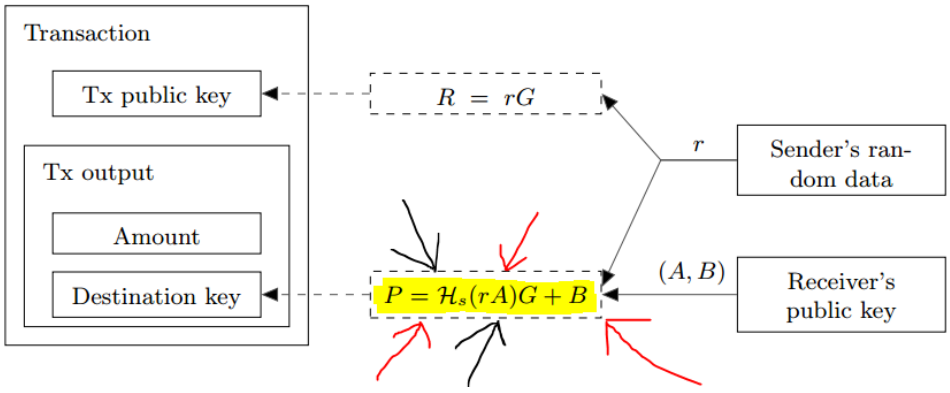
In case it isn't clear why Alice's $D$ equals Bob's $D'$, here's an example:

1. Use base point $G$.
2. Use $a = 3$; $A = 3G$.
3. Use $b = 4$; $B = 4G$.
4. $a * b = 12$.
5. "Alice's" $D = aB = 3B = 3 * 4G = 12G$.
6. "Bob's" $D' = bA = 4A = 4 * 3G = 12G$!

Note that $D$ has a corresponding scalar $d$ (12 in the above example), that no one knows. We only know it in this case because we know both $a$ and $b$! This isn't particularly useful information, but is the kind of thing I find interesting.

### 3.3.2 Stealth Addresses

Now that you know about ECDH, let's move on to how **dual-key stealth addresses** actually work! Again from the Cryptonote whitepaper, we get:




As I've helpfully highlighted and pointed out here, we're looking at $P = H_s(rA) \cdot G + B$.

**Dual-key** simple refers to the pairs of spend/view keys, which allows "decoding" (or removing the unlinkability if you will) stealth addresses *without simultaneously allowing them to be spent.*

Now, ignore all that. Let's back up.

Alice wants to send a payment to Bob. Alice's **private spend key** is $z$, and her **private view key** is $\mathbf{y}$. Her public address is then $(Z, Y)$ in holy whitepaper order. I've helpfully used letters that aren't referenced above, because *Alice's keys aren't used at all.*

Bob's **public address** is $(A, B)$. Remember from the previous article that the whitepaper helpfully uses $A$ as the public view key and $B$ as the public spend key. Presumably Bob's **private keys** would be $(a, b)$, but Alice (our current perspective) doesn't know them.

The final piece needed before "building" our first stealth address is $r$ and $R$. $r$ is a new random scalar chosen by Alice for the express purpose of creating a stealth address for Bob. $R$ is the corresponding curve point for $rG$. $r$ is not given to anyone and may be destroyed after use, unless Alice wants to later prove to a 3rd party that she paid Bob. $R$, however, is added to the transaction for everyone to see. A new $r$ should be chosen for every single transaction (reusing r to send to the same recipient would result in a stealth address collision!).

Here is an example of R at chainradar.com:


Now that we have our key definitions out of the way, let's create a stealth address! I think walking through the process is the easiest way to understand.

$$\mathbf{P} = \mathbf{H_s}(\mathbf{rA}) \cdot \mathbf{G} + \mathbf{B}$$

Referenced above we have:

1. $P$ – the final **stealth address** (one-time output key, the destination where funds will actually be sent);
2. $H_s$* – a hashing algorithm that returns a scalar (i.e., the hash output is interpreted as an integer and reduced modulo $l$);
3. $r$ – the new random scalar Alice chose for this transaction;
4. $A$ – Bob's **public view key**;
5. $G$ – the standard Ed25519 base point;
6. $B$ – Bob's **public spend key**.

[*Note: the hashing algorithm of choice for Monero and other Cryptonote coins is keccak-256. This article's focus is not on hashing algorithms, so if you want to learn more about them please see the linked article. Wikipedia, however, has a good short list of the properties we want:

1. it is quick to compute the hash value for any given message
2. it is infeasible to generate a message from its hash value except by trying all possible messages
3. a small change to a message should change the hash value so extensively that the new hash value appears **uncorrelated** with the old hash value

4. it is infeasible to find two different messages with the same hash value

#1 is obvious. #2 is the one-way property. #3 is very interesting, as "uncorrelated" is quite similar to our magic word, **unlinkable**. #4, collision resistance, is very important because a bad algorithm could significantly "improve" the chance of a stealth address collision (versus the $2^{126}$ discussed above).

I would add to this list:

1. any length input;
2. fixed-length output;
3. output cannot be predicted or chosen in advance (preimage-resistance) – this is related to #2 above.]

Whew, that got long.

**Alright, so let's actually create a stealth address!**

1. Alice does ECDH with her randomly-chosen r and Bob's public view key, $A$. Let's call this point **D**. No one other than Alice or Bob can compute **D** (see the discussion above on ECDH).
2. Alice uses $D$ to generate a new scalar; we'll call it $f$. $f = H_s(D)$. Yes I like naming things. This is the step that actually **causes unlinkability** between Bob's outputs (remember #3 above – more on this later)!
3. Alice computes $F = fG$.
4. Alice computes $P = F + B$ (Bob's public spend key).
5. $P$ is the stealth destination!

**Now let's look at Bob's perspective:**

1. Bob receives a transaction; he wants to check if it belongs to him.
2. Bob retrieves $R$, which Alice has helpfully attached to the transaction.
3. Bob computes $D'$. Note Bob doesn't (yet) know if $D'$ is equal to $D$. $D' = aR$.
4. Bob computes $f' = H_s(D')$.
5. Bob computes $F' = f'G$.
6. Bob computes $P' = F' + B$.
7. Bob checks if $P'$ is equal to $P$, which was included in the transaction as the destination. If yes, Bob realizes he's been paid and does some additional steps (below). If no, Bob ignores the transaction.

**Some notes:**

1. Computing $D$ and $D'$ requires secret data: either $r$ (Alice) or $a$ (Bob). Thus, external observers are prevented from proceeding past Alice's step 1. Furthermore, because $r$ is randomly chosen, even if the observer suspects Alice is sending to Bob's public address (which the observer knows), due to the ECDLP they still can't link this address to $P$ without knowledge of $r$ or $a$ (or pedantically the later steps' values, namely $D$ and $f$).

2. You may have noticed that this scheme only gives Alice one output for Bob per $r$, but with auto-denomination Monero and other Cryptonote coins have many outputs per transaction. To get different stealth addresses for each output, Alice (and Bob) append an "output index" (an output's position in the transaction: 0, 1, 2, etc.) to $D$ before hashing it to create the secret shared scalar f. This is a bit of a clarification to Alice's step 2 on **unlinkability**. That is, while the shared secret $D$ is already unlinkable to observers, appending an output index allows "unlimited" additional unlinkable outputs to be created from one shared secret (see point 3 in the hash section).

3. Back to the **dual-key** concept, Bob (or someone working on his behalf with knowledge of $a$ and $B$) can "scan" for and detect/link outputs without knowledge of $b$, which is required below to actually spend that output. The whitepaper calls $(a, B)$ the "tracking key".

4. It is possible to do a non-dual-key stealth addressing scheme, but you must make one of two trade-offs. You can either:

   - use the concept in the whitepaper called a **truncated address**, which means the view key pair is publicly known and all incoming transactions can be linked ($a = Hs(B)$); or
   - forego a view key pair entirely, which means scanning requires spending ability ($P = Hs(rB) \cdot G + B$)).

**Bob's Additional Steps**

So, Bob has determined some outputs in a transaction belong to him. Now what does he do? I can imagine two things he *wants* to do: check if this output has already been spent, and (later) actually spend the output. To do either of these things, Bob must first compute the secret key associated with that output (the one-time secret key, $x$).

1. Bob recomputes $f' = Hs(D')$ (as above).

2. Bob derives $x = f' + b$ ($b$ is Bob's private spend key). The "neat" thing is that $P = xG$! Adding scalars (or points) together preserves linearity. $P = xG = (f' + b) \cdot G = F' + B$

3. To check if $P$ is spent, Bob computes its "key image" and queries the blockchain to see if it is marked as spent. Key image $I = xH_p(P)$. This will be better explained in the ring signature article. *Fear not!*

4. To spend $P$, Bob needs to sign a new transaction with $x$ (also detailed in the ring signature article).

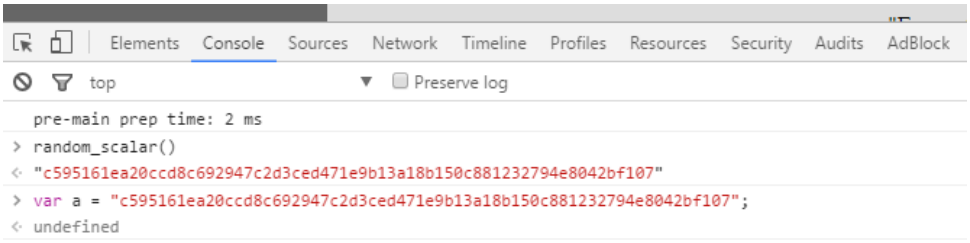
**[Skip the next section if you hate fun.]**

### 3.3.3 FUN

**Now, let's do a "fun" exercise with real values for illustration.** *Great...*

I'll be using functions that are available in Javascript here. Doing so allows the curious and discerning reader (*that's you, presumably*) to easily reproduce the results with only a web browser. If you go to the link above and open your browser's console (right-click the page->Inspect, then Console tab), you can enter or copy/paste all the commands below to "see it in action". All letters and step numbers match those above. The scalars b and r below were randomly generated with `random_scalar()`; (you obviously can't randomly generate your own if you want to reproduce my results!). You can see the contents of a variable by just typing its name and pressing . "//" is a comment in Javascript.

Here is the Chrome console:




### Preliminary

```
//Bob's private spend key
 var b = "c595161ea20ccd8c692947c2d3ced471e9b13a18b150c881232794e8042bf107";

//Bob's private view key (deterministic derivation);
//a = "fadf3558b700b88936113be1e5342245bd68a6b1deeb496000c4148ad4b61f02"
var a = hash_to_scalar(b);

//Bob's public spend key, this function multiplies base G by its input;
//B = "3bcb82eecc13739b463b386fc1ed991386a046b478bf4864673ca0a229c3cec1"
var B = sec_key_to_pub(b);

//Bob's public view key;
//A = "6bb8297dc3b54407ac78ffa4efa4afbe5f1806e5e41aa56ae98c2fe53032bb4b"
var A = sec_key_to_pub(a);

//returns Bob's public address (for curiosity only),
//"43tXwm6UNNvSyMdHU4Jfeg4GRgU7KEVAfHo3B5RrXYMjZMRaowr68y12HSo14wv2qcYqqpG1U5AHrJtBdFHKPDE
    A9UxK6Hy"
pubkeys_to_string(B,A);

var r = "c91ae3053f640fcad393fb6c74ad9f064c25314c8993c5545306154e070b1f0f";

//R = "396fc23bc389046b214087a9522c0fbd673d2f3f00ab9768f35fa52f953fef22"
var R = sec_key_to_pub(r);
```

### Alice

```
//ECDH, rA; D = "a1d198629fadc698b48f33dc2e280301679ab2c75a76974fd185ba66ab8418cc"
var D = generate_key_derivation(\(A\), r);

//0 is the output index; the standard method combines these last few steps into one, but I
    split them for clarity;
// f = "bf1d230a09bfdb0bc7fe04cddf8c1635d0ebaaf159ef85dc408ae60879752509"
var f = derivation_to_scalar(D, 0);

//F = "3e4b39c5b5110d6fbdb77fbcd203709c19fefd28c982a86bda3f3d35fc099738"
var F = sec_key_to_pub(f);

//``ge'' means group element; this function adds two points together.
//P = "6cabaac48d3b9043525a703e9e5feb72132f69ea6deca9b4acf9228beb74cd8f"
var P = ge_add(F,B);
```

**Bob**

```
N/A
N/A
var D1 = generate_key_derivation(R, a); //D1 = same as above!
var f1 = derivation_to_scalar(D1, 0); //f1 = same as above!
var F1 = sec_key_to_pub(f1); //F1 = same as above!
var P1 = ge_add(F1,B); //P1 = same as above!
```

**Bob's additional steps**

```
N/A

//"sc" means scalar; this function adds two scalars together.
// x = "97df43cb906896405a8b54ecd4610c92b99de5090b404e5e64b17af17da01601". Now for fun enter sec_key_to_pub(x); and compare with P.
var x = sc_add(f1,b);

//This combines "hash_to_ec" (Hp, hash to a curve point) and a scalar multiplication of that new point by x.
// I = "2ba7ee37314d4a1edbeef727f49099c79d55797570cb1206ee2685c94b6550b1"
// For fun -- spent status
var I = generate_key_image_2(P, x);9
N/A
```

Whew!

**[End skip.]**


tl;dr (*You get one this time!*), stealth addressing allows senders to create "unlimited" one-time destination addresses on behalf of the recipient (without any interaction). They can only be recovered and spent by the recipient and can't be publicly linked to each other or the standard address from which they were derived.

Until next time!

## 3.4 Borromean Ring Signature 原理

By Hao Xu.

### 3.4.1 签名

假设签名者 Alice 有一个公钥的集合 $\{P_{i,j}|0 \le i \le n-1, 0 \le j \le m_i - 1\}$。其中 $n$ 是环的个数，$m_i$ 是第 $i$ 个环的元素个数。Alice 希望用 $n$ 个 key $\{P_{i,a_i}\}$ 对应的私钥 $x_i$ 去产生一个环签名，其中 $a_i$ 由 Alice 挑选，作为环的起点，验证者并不知道。

Alice 签名步骤如下：

1. 对消息 $m$ 进行哈希得到 $M$，并统计所有公钥。
2. 对于每一个 $0 \le i \le n-1$,
   (a) 随机选择一个标量 $k_i$.
   (b) 设置 $e_{i,a_i+1} = H(M||k_iG||i||a_i)$.
   (c) 对于每个 $a_i + 1 \le j < m_i - 1$, 随机选取 $s_{i,j}$ 的值，并计算
   $$e_{i,j+1} = H(M||s_{i,j}G - e_{i,j}P_{i,j}||i||j).$$
   至此计算了每个环中编号大于 $a_i$ 的所有点的 $e$ 值和 $s$ 值。
3. 对于每个环，选取编号最大的点的 $s$ 和 $e$ 值，分别为 $s_{i,m_i-1}$ 和 $e_{i,m_i-1}$。并计算 $e_0$，公式如下：
   $$e_0 = H(s_{0,m_0-1}G - e_{0,m_0-1}P_{0,0}|| \dots ||s_{n-1,m_{n-1}-1}G - e_{n-1,m_{n-1}-1}P_{n-1,m_{n-1}-1})$$
   由公式可知，$e_0$ 值取决于 $n$ 个点的 $s$ 和 $e$ 的值。
4. 对每个 $0 \le i \le n-1$:
   (a) 对于每一个 $0 \le j < a_i - 1$, 随机选择 $s_{i,j}$ 的值，并计算
   $$e_{i,j+1} = H(M||s_{i,j}G - e_{i,j}P_{i,j}||i||j).$$
   其中 $e_{i,0}$ 即为 $e_0$。这一步计算了每个环中编号小于 $a_i$ 的所有点的 $e$ 值和 $s$ 值，以及 $a_i$ 的 $e$ 值。
   (b) 计算 $a_i$ 的 $s$ 值如下：$s_{i,a_i} = k_i + x_i e_{i,a_i}$.

最后生成的环签名如下：
$$\sigma = \{e_0, s_{i,j} : 0 \le i \le n, 0 \le j \le m_i\}$$

### 3.4.2 验签

假设验证者的已知信息为消息 $m$, 公钥集 $\{P_{i,j}|0 \le i \le n-1, 0 \le j \le m_i - 1\}$ 和签名 $\sigma$。
验证者验证签名过程如下：

1. 对消息 $m$ 进行哈希得到 $M$.
2. 对于每个 $0 \le i \le n-1$ 和 $0 \le j \le m_j - 1$，计算 $R_{i,j+1} = s_{i,j}G + e_{i,j}P_{i,j}$ 和 $e_{i,j+1} = H(M||R_{i,j+1}||i||j)$.
3. 计算 $e'_0 = H(R_{0,m_0-1}|| \dots ||R_{n-1,m_{n-1}-1})$。并比较 $e'_0 \stackrel{?}{=} e_0$，若相等，则验证通过。

以上签名过程的示意图如 Figure 3.2.

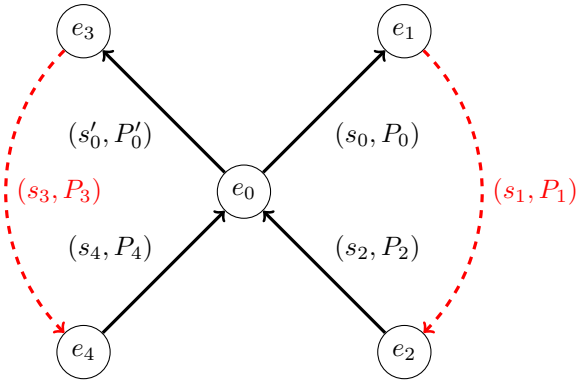



Figure 3.2: A Borromean ring signature for $(P_0|P_1|P_2)\&(P'_0|P_3|P_4)$

## 3.5 Mix-ins Construction

By Xiangmin Li on July 30th, 2017.

### 3.5.1 Notations

- $n$: number of mix-in requested
- $pk_i$: public keys of the output indexed by the global output index $i$
- $C_i$: commitment of the output indexed by the global output index $i$
- *unlocked*: means the outputs are available for spending
- $x$: the number of RCT outputs, including those spent. Hence, it's also the upper index limit of global indices of these outputs
- $y$: the number of unlocked RCT outputs among all RCT ones, including those spent. Hence, it' s also the upper index limit of global indices of these unlocked outputs
- $z$: the number of recent RCT outputs among $y$ unlocked ones. Hence, the range of indices for recent RCT outputs should be $[y-z, y]$
- $\Delta w_c$: a time window of recent cut off, i.e., the time interval of interest is $[t - \Delta w_c, t]$ given $t$ is a referenced moment
- $\Delta w_1$: the unlocked time window for money
- $\Delta w_2$: the default spendable age of the transaction
- $\pi$: the global index of the real output

### 3.5.2 About the global output index

Similar to the CryptoNote protocol, outputs in Monero are organized into groups according to their amounts, and a global output index represents the index of a given output in its corresponding group. This reduces the size of the ring signature by only storing those indices of outputs in the signature instead of the actual public keys.

After the activation of RingCT, the denomination of outputs is no longer necessary, and all the RingCT outputs (whose amounts are hidden) are stored in the same group (associated with the amount of 0 XMR by convention). There are 1491097 RingCT outputs as of now, and this transaction generated the two latest outputs at indices of 1491095 and 1491096.

For convenience, all the statement henceforth will only refer to RingCTs.

### 3.5.3 Workflow

There's no check for double spending in the mix-ins selection procedure. And the detail goes as following sequence diagram as Figure 3.3.

## 3.6 RCT implementation

By Xiangmin Li, at 2017-08-11.

### 3.6.1 Overview

Let $m$, $\{l_j\}_{j=1}^m$ be parameters of the Ring-CT scheme, where

- $m$ is the number of input coins
- $\{l_j\}$ is a parameter governing the size of the anonymity set hiding the $j$-th input coin $\mathcal{S}_j$

There are 2 kinds of Ring-CT in monero project, called the simple Ring-CT ($m > 1$) and the full Ring-CT ($m = 1$), where the full version can only be used in case of one ring. These 2 versions of Ring-CT are detailed in following subsections respectively.

### 3.6.2 Simple Ring-CT

**Preparation of the transaction key** Generate a seckey-pubkey pair randomly, $(r, R = r \cdot G)$ for the transaction to conduct.

**Preparation of the output coins** For each coins

$$coin_k^{(out)} = \left(P_k^{(out)}, cn_k^{(out)}, \Pi_k^{(range)}\right)$$

to send, compute

- destination address $P_k^{(out)} = H_s(r \cdot A_k) \cdot G + B_k$, where $A_k$ and $B_k$ are respectively, the public viewing key and public spending key for the targeted payee, and $H_s$ means hashing its input to a scalar over ed25519.
- output commitment $cn_k^{(out)} = y_k \cdot G + b_k \cdot H$, where $y_k$ is the amount key and $b_k$ is the amount
- $\Pi_k^{(range)}$ is range proof for amount $b_k$

**Mixing Input Coin with Anonymity Set** Parse the input coins as $\mathcal{S} = \{\mathcal{S}_j = (P_j, cn_j)\}_{j=1}^m$, where $cn_j = x_j \cdot G + a_j \cdot H$ commits the denomination $a_j$ of $\mathcal{S}_j$. For each $\mathcal{S}_j$, choose arbitrary $l_j - 1$ coins as mix-ins, thus forming a coin vector. Shuffle coins in $j$-th vector to get the mixed coin set $M_j = [(P_{j,1}, cn_{j,1}), (P_{j,2}, cn_{j,2}), \ldots, (P_{j,l_j}, cn_{j,l_j})]$ arranged as a column vector.

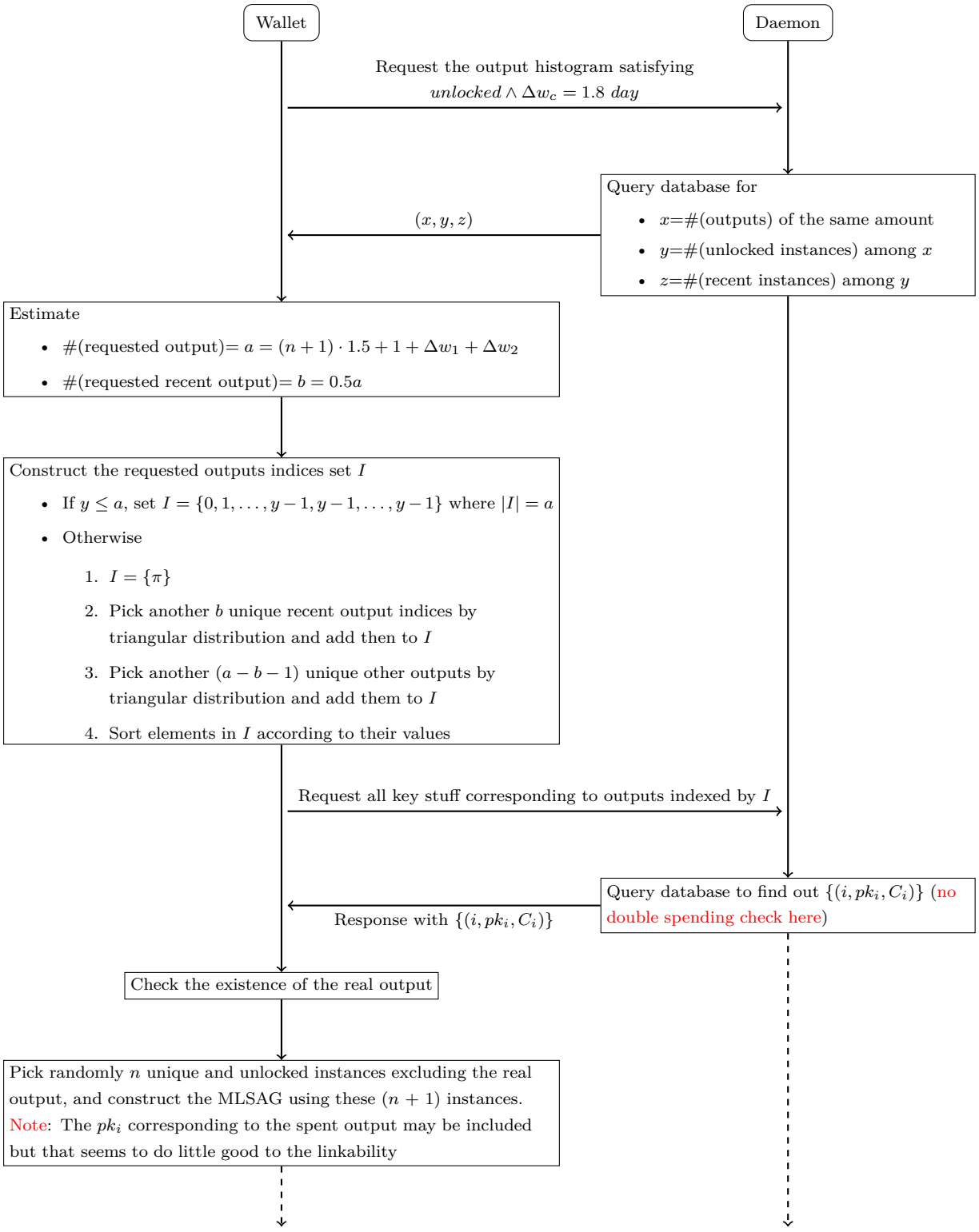



Figure 3.3: Mix-ins selection procedure

**Preparation of Ring $\mathcal{L}_j$ for $j = 1, \ldots, m$.** For each $j = 1, 2, \ldots, (m-1)$, there is exactly one $\pi_j \in \{1, 2, \ldots, l_j\}$ such that one coin $(P_{j,\pi_j}, cn_{j,\pi_j}) \in M_j$ is $\mathcal{S}_j$. Make a pseudo commitment $cn_j^* = x_j^* \cdot G + a_j \cdot H$ (i.e., a commitment on the same amount as the real input coin $\mathcal{S}_j$, which is called `pseudoOuts` in the codebase). And for $\mathcal{L}_m$, make its pseudo commitment as

$$cn_m^* = x_m^* \cdot G + a_m \cdot H = \left( \sum y_k - \sum_{j=1}^{m-1} x_j^* \right) \cdot G + a_m \cdot H$$

For $j = 1, 2, \ldots, m$, parse ring as

$$\mathcal{L}_j = \begin{bmatrix} P_{j,1} & P_{j,2} & \cdots & P_{j,l_j} \\ (cn_{j,1} - cn_j^*) & (cn_{j,2} - cn_j^*) & \cdots & (cn_{j,l_j} - cn_j^*) \end{bmatrix}$$

We know the secret key for column $[P_{j,\pi_j}, cn_{j,\pi_j}]^T$ as $[sk_j, (x_j - x_j^*)]^T$

**Generation of MLSAG** For each $\mathcal{L}_j$, create an MLSAG on message

$$M = H_s \left( H_s(\{\mathcal{S}_j\}_{j=1}^m, \{P_k^{(out)}\}), \{cn_j^*\}_{j=1}^m, \{cn_k^{(out)}\} \right)$$

(Other relevant but not so important parameters are left out for brevity here). The output of signing is $(\mathcal{SPK}_j, I_j)$, where $\mathcal{SPK}_j$ is the ring signature on $\mathcal{L}_j$ and $I_j$ is the key image (a.k.a, the serial number) of the input key pair $(sk_j, P_j = sk_j \cdot G)$. Parse the final signature $\sigma$ on the ring set $\{\mathcal{L}_j\}_{j=1}^m$ as

$$(\mathcal{SPK}_1, I_1, cn_1^*, \ldots, \mathcal{SPK}_m, I_m, cn_m^*)$$

The signature $\sigma$, transaction public key $R$ together with the set of output amount commitment $\{cn_k^{(out)}\}$ and their range proof $\{\Pi_k^{(range)}\}$ forms the proof that the transaction is conducted correctly. The payer encrypts the output amount key $y_k$ and denomination $b_k$ with the shared secret key $ss = H_s(r \cdot A_k)$ as follows

$$\begin{aligned} y_k^* &= y_k + H_s(ss) \\ b_k^* &= b_k + H_s(H_s(ss)) \end{aligned}$$

and put $\{(y_k^*, b_k^*)\}$ in the transaction.

**Verification** Re-create

$$\mathcal{L}_j = \begin{bmatrix} P_{j,1} & P_{j,2} & \cdots & P_{j,l_j} \\ (cn_{j,1} - cn_j^*) & (cn_{j,2} - cn_j^*) & \cdots & (cn_{j,l_j} - cn_j^*) \end{bmatrix}$$

using $cn_j^*$. Then invoke the verification algorithm of the MLSAG to verify the $\mathcal{SPK}_j$. And the amount commitment is checked as verifying the following equality

$$\sum cn_k^{(out)} + b_0 \cdot H = \sum_{j=1}^m cn_j^*$$

where $b_0$ is the transaction fee. Also, check double spending and the range proof $\Pi_k^{(range)}$.

### 3.6.3 Full Ring-CT

**Preparation of the transaction key** Generate a seckey-pubkey pair randomly, $(r, R = r \cdot G)$ for the transaction to conduct.

**Preparation of the output coins** For each coins $coin_k^{(out)} = (P_k^{(out)}, cn_k^{(out)}, \Pi_k^{(range)})$ to send, compute

- destination address $P_k^{(out)} = H_s(r \cdot A_k) \cdot G + B_k$, where $A_k$ and $B_k$ are respectively, the public viewing key and public spending key for the targeted payee
- output commitment $cn_k^{(out)} = y_k \cdot G + b_k \cdot H$, where $y_k$ is the amount key and $b_k$ is the amount
- $\Pi_k^{(range)}$ is range proof for amount $b_k$

**Mixing Input Coin with Anonymity Set** Parse the only input coin as $\mathcal{S}_0 = (P, cn)$, where $cn = x \cdot G + a \cdot H$ commits the denomination $a$ of $\mathcal{S}_0$. Choose arbitrary $(l-1)$ coins as mix-ins, thus forming a coin vector. Shuffle coins in the vector to get the mixed coin vector $M_0 = [(P_1, cn_1), (P_2, cn_2), \ldots, (P_l, cn_l)]$ arranged as a column vector.

**Preparation of Ring $\mathcal{L}_0$** There is exactly one $\pi \in 1, 2, \ldots, l$ such that one coin $(P_\pi, cn_\pi) \in M_0$ is $\mathcal{S}_0$. Then parse the ring as

$$\mathcal{L}_0 = \begin{bmatrix} P_1 & \cdots & P_l \\ \left(cn_1 - b_0 H - \sum cn_k^{(out)}\right) & \cdots & \left(cn_l - b_0 H - \sum cn_k^{(out)}\right) \end{bmatrix}$$

where $b_0$ is the transaction fee. We know the secret key for column $[P_\pi, cn_\pi]^T$ as $[sk, (x - \sum y_k)]^T$

**Generation of MLSAG** Create an MLSAG on message based on ring $\mathcal{L}_0$

$$M = H_s\left(H_s(\mathcal{S}_0, \{P_k^{(out)}\}), cn, \{cn_k^{(out)}\}\right)$$

(Other relevant but not so important parameters are left out for brevity here). The output of signing is $(\mathcal{SPK}_0, I_0)$, where $\mathcal{SPK}_0$ is the ring signature on $\mathcal{L}_0$ and $I_0$ is the key image (a.k.a, the serial number) of the input key pair $(sk, P = sk \cdot G)$. Parse the final signature $\sigma$ on the ring $\mathcal{L}_0$ as $(\mathcal{SPK}_0, I_0)$.

The signature $\sigma$, transaction public key $R$ together with the set of range commitment $\{cn_k^{(out)}\}$ forms the proof that the transaction is conducted correctly. The payer encrypts the output amount key $y_k$ and denomination $b_k$ with the shared secret key $ss = H_s(r \cdot A_k)$ as follows

$$\begin{aligned} y_k^* &= y_k + H_s(ss) \\ b_k^* &= b_k + H_s(H_s(ss)) \end{aligned}$$

and put $\{(y_k^*, b_k^*)\}$ in the transaction.

**Verification** Re-create

$$\mathcal{L}_0 = \begin{bmatrix} P_1 & \cdots & P_l \\ \left(cn_1 - b_0 H - \sum cn_k^{(out)}\right) & \cdots & \left(cn_l - b_0 H - \sum cn_k^{(out)}\right) \end{bmatrix}$$

Then invoke the verification algorithm of the MLSAG to verify the $\mathcal{SPK}_0$. Also, check double spending and the range proof $\Pi_k^{(range)}$.

## 3.7 StackExchange Series

### 3.7.1 Important contributors to monero project

- vtnerd

### 3.7.2 About Commitment

- How are outPk, mask and amount fields created when spending RignCT coinbase txs
- What are 3 types of Ring CT transactions answered by vtnerd