Monero's Building Blocks
Part 9 of 10 – *RingCT and anatomy of Monero transactions*

Bassam El Khoury Seguias

BTC: 3FcVvBZwTUkUrcqJd16RcjR42qT2tDWHWn

ETH: 0xb79Fb9194C8Cc6221368bb70976e18609Ab9AcA8

April 30, 2018

1 Introduction

In part 7 we introduced the **MLSAG ring signature** scheme. Among other things, it safeguarded the anonymity of the signer. In part 8 we discussed the notions of **Pedersen Commitments** and **Confidential Transactions**. They were used to mask transaction amounts without compromising the proper bookkeeping of balances on the network.

It turns out that combining both concepts in a single mathematical construct requires additional work. In the first section, we explain why outright combination of the aforementioned concepts fails to preserve the anonymity of the sender.

In the second section we remedy the situation by introducing the notion of a non-zero commitment. This will form the basis of Monero's ringCT scheme.

The last section goes over the mechanics of how a Monero transaction is created and includes references to relevant parts of the code base. We introduce two variants of ringCT, namely ringCT Type Full and ringCT Type Simple. We finally conclude with a breakdown of the components of a real-life Monero transaction.

2 A problem with preserving anonymity

A Monero transaction is a mathematical construct that is cryptographically signed. It details what unspent transaction outputs (UTXOs) the sender wants to use to conduct his transfer. In essence, a UTXO is an output associated with a previous blockchain transaction that hasn't been spent yet. It can be subsequently used as an input to a future transaction. In addition, a Monero transaction encapsulates details about the recipients of the funds including the amount to be transferred to each recipient.

Consider the following hypothetical Monero transaction:

- The sender uses two of her UTXOs with respective amounts $(a_{in})_1 = XMR\ 2$ and $(a_{in})_2 = XMR\ 4$ (i.e., units of Monero currency).
- The sender transfers funds to three recipients including a transaction fee to the miners, funds destined to a counterparty, and change returned to the sender. Suppose the output amounts are respectively $txfee \equiv (a_{out})_1 = XMR\ 1$, $(a_{out})_2 = XMR\ 4$, $change \equiv (a_{out})_3 = XMR\ 1$.
- All input and output amounts are replaced with a corresponding Pederson Commitment to hide the original value.

Suppose that we would also like to conceal the identity of the sender. This is tantamount to hiding the origin of the funds (i.e., hiding the sender's UTXOs). Each UTXO is associated with a **"one-time public key"** and a corresponding **"one-time private key"** (the details will be explained in part 10 when we discuss the **stealth addressing system**). The sender embeds each UTXO in a set of n other UTXOs. In this example we let $n = 5$. The following is a representation of this scenario where each UTXO is identified by its transaction hash, commonly known as **transaction id** [2]

- First set of 5 UTXOs

  $\{$ $00 : 555...62f$ (1^{st} UTXO of ring member #1)
  $\{$ $01 : 0d3...34e$ (1^{st} UTXO of ring member #2)
  $\{$ $02 : 6eb...8b9$ (1^{st} UTXO of ring member $\#3 \equiv \pi$ (index of the signer))
  $\{$ $03 : 7d4...32a$ (1^{st} UTXO of ring member #4)
  $\{$ $04 : 4a7...9fe$ (1^{st} UTXO of ring member #5)

- Second set of 5 UTXOs

  $\{$ $00 : a96...54f$ (2^{nd} UTXO of ring member #1)
  $\{$ $01 : 783...a9b$ (2^{nd} UTXO of ring member #2)
  $\{$ $02 : 328...be3$ (2^{nd} UTXO of ring member $\#3 \equiv \pi$ (index of the signer))
  $\{$ $03 : 150...6e9$ (2^{nd} UTXO of ring member #4)
  $\{$ $04 : 754...3df$ (2^{nd} UTXO of ring member #5)

To achieve the above, we can build an MLSAG ring signature where:

- The **"one-time private keys"** of all UTXOs used by the sender are grouped together to form his private key vector $[x_\pi^1\ x_\pi^2]^T$. Cleary, the private key vector will have an associated public key vector $[y_\pi^1\ y_\pi^2]^T$ where $y_\pi^j = x_\pi^j \otimes G,\ j \in \{1, 2\}$.
- The remaining 4 ring members will be associated with four additional public key vectors. Each vector consists of a pair of UTXOs that are pairwise different and different from the ones used by the sender. The total number of ring members, not including the sender, is known as the **mixin count** in Monero. Our hypothetical example has a **mixin count** of 4.

This set-up can be summarized in a public key matrix given by:

$$PK = \begin{bmatrix} y_1^1 & y_2^1 & y_{\pi\equiv 3}^1 & y_4^1 & y_5^1 \\ y_1^2 & y_2^2 & y_{\pi\equiv 3}^2 & y_4^2 & y_5^2 \end{bmatrix}$$

As noted earlier, the transaction amounts associated with each UTXO are concealed using Pedersen Commitments. Following the logic outlined in part 8, the network will check if $\sum_{i=1}^{2}[(C_{in})_i]_k = \sum_{j=1}^{3}(C_{out})_j$ for some $k \in \{1, 2, 3, 4, 5\}$ denoting the index of a ring member. Once it identifies a ring member k whose UTXO pair satisfies this equality, it gets the assurance that the amounts balance out without knowing what the amounts are.

Note that the index of both UTXOs in a given pair must be the same. For example, the sender's first UTXO appears in the 3rd position in the first UTXO set and also in the 3rd position in the second set. This is dictated by the construction of the private key and public key vectors in MLSAG.

However, by finding which pair of UTXOs satisfy the equation, the network would have also discovered the index of the signer. This is because it is extremely unlikely to select different ring members (i.e., pairs of UTXOs) such that the sum of their Pedersen Commitments (i.e., $\sum_{i=1}^{2}[(C_{in})_i]_k$) matches that of the sender (i.e., $\sum_{i=1}^{2}[(C_{in})_i]_\pi$). Consequently, only the Pedersen Commitments of the sender will satisfy the equation. By figuring out the index of the signer, the anonymity of the MLSAG gets jeopardized. In order to address this problem, we introduce an amendment to the MLSAG public key matrix.

3 Ring Confidential Transaction (RingCT)

Recall the following set-up introduced in part 8:

- $\forall i \in \{1, ..., m\}$, let $(C_{in})_i = ((x_{in})_i \otimes G) \oplus ((a_{in})_i \otimes H)$ be the Pedersen Commitment associated with amount $(a_{in})_i$ with blinding factor $(x_{in})_i$ randomly chosen in $\mathbb{F}_l$.
- Let $(a_{out})_t \equiv txfee$ be the miner's transaction fee and let $(C_{out})_t = (a_{out})_j \otimes H$ be the Pedersen Commitment associated with $txfee$. The blinding factor $(x_{out})_t$ is deliberatly chosen to be 0 (i.e., the identity element of $\mathbb{F}_l$).
- $\forall j \in \{1, ..., t-1\}$, let $(C_{out})_j = ((x_{out})_j \otimes G) \oplus ((a_{out})_j \otimes H)$ be the Pedersen Commitment associated with amount $(a_{out})_j$ with blinding factor $(x_{out})_j$ randomly chosen in $\mathbb{F}_l$. We additionaly require that $\sum_{i=1}^{m}(x_{in})_i - \sum_{j=1}^{t-1}(x_{out})_j = 0 \pmod{l}$.

By ensuring that:

1. $\sum_{i=1}^{m}(x_{in})_i - \sum_{j=1}^{t-1}(x_{out})_j = 0 \pmod{l}$, and
2. $\forall i \in \{1, ..., m\},\ \forall j \in \{1, ...t\}$ the amounts $(a_{in})_i$ and $(a_{out})_j$ remain confined to a pre-defined range $[0, 2^r] \subset \mathbb{F}_l$ (refer to part 8 for more information about the choice of r).

we got the following equivalence:

$$\sum_{i=1}^{m}(C_{in})_i \ \ominus\ \sum_{j=1}^{t}(C_{out})_j = 0$$

$$\iff \sum_{i=1}^{m}(a_{in})_i - \sum_{j=1}^{t}(a_{out})_j = 0$$

It is important to observe that the amounts balance out in actuality and not in the more relaxed modulo l sense. This is due to the constraint we imposed on all transaction amounts being confined to the $[0, 2^r]$ range. If this were not the case, one would be able to create or destroy Monero currency while still maintaining a balanced equation. To see this, suppose transaction amounts can take on any value in $\mathbb{F}_l$ instead of being restricted to $[0, 2^r]$. Moreover, suppose that the sender uses three UTXOs (i.e., $m = 3$) with $(a_{in})_1 = XMR\ (l-4)$, $(a_{in})_2 = XMR\ 3$, and $(a_{in})_3 = XMR\ 5$. There are two outputs (i.e., $t = 2$) with $(a_{out})_1 = XMR\ 3$, and $(a_{out})_2 = XMR\ 1$.

$$\text{Clearly, } \sum_{i=1}^{m}(a_{in})_i - \sum_{j=1}^{t}(a_{out})_j = l \neq 0.$$

$$\text{However, } \sum_{i=1}^{m}(a_{in})_i - \sum_{j=1}^{t}(a_{out})_j = 0 \pmod{l}.$$

If this transaction gets approved by the network, we would have effectively destroyed l units of currency. Conversely, exchanging the input and output values would allow the creation of l units of currency out of thin air. This example demonstrates the importance of having a balanced equation independent of modulo l arithmetic. By confining all transaction amounts to the $[0, 2^r]$ range, we ensure that this is the case. To prove that a transaction amount lies in a certain range, Monero makes use of the **Borromean signature** construct. A more size-efficient alternative to Borromean signatures that is currently deployed on Monero's testnet is **Bulletproof**. Bulletproof performs a range proof while potentially decreasing a Monero transaction size (and hence transaction fees) by up to 80%. Range proofs including the Borromean and Bulletproof constructs deserve an article on their own. We might dedicate an appendix to expain how they work in the future. For the time being, the interested reader can consult [3] and [1].

The problem encountered in section 2 was due to the high likelihood that only ring member $\pi,\ 1 \leq \pi \leq 5$ had UTXOs whose Pedersen Commitments satisfy the equation $\sum_{i=1}^{2}[(C_{in})_i]_\pi \ \ominus\ \sum_{j=1}^{3}(C_{out})_j = 0$. (Note that we explicitly mention the index π in $\sum_{i=1}^{2}[(C_{in})_i]_\pi$ to highlight that these specific Pedersen Commitments are the ones associated with the sender's UTXOs. Other ring members have different UTXOs and hence different commitments). The culprit is the value 0 which gave away the index π of the sender. To remedy this shortcoming, we relax the condition: instead of requiring that $\sum_{i=1}^{m}[(x_{in})_i]_\pi - \sum_{j=1}^{t-1}(x_{out})_j = 0 \pmod{l}$, we let it take on any scalar value $z \in \mathbb{F}_l^*$, **as long as z is only known to the sender** π. We highlight that the blinding factors $[(x_{in})_i]_\pi$ are the ones associated with π's i^{th} UTXO. Carrying over the notation from part 8, we get the following equalities:

$$\sum_{i=1}^{m}[(C_{in})_i]_\pi \ \ominus\ \sum_{j=1}^{t}(C_{out})_j$$

$$= \sum_{i=1}^{m}[(C_{in})_i]_\pi \ \ominus\ \sum_{j=1}^{t-1}(C_{out})_j \ \ominus\ (txfee \otimes H) \text{ (by definition of } txfee \text{ and } (C_{out})_t)$$

$$= \sum_{i=1}^{m}\{([(x_{in})_i]_\pi \otimes G) \oplus ([(a_{in})_i]_\pi \otimes H)\} \ \ominus\ \sum_{j=1}^{t-1}\{((x_{out})_j \otimes G) \oplus ((a_{out})_j \otimes H)\} \ \ominus\ (txfee \otimes H)$$

$$= \sum_{i=1}^{m} k([(x_{in})_i]_\pi, [(a_{in})_i]_\pi) \ \ominus\ \sum_{j=1}^{t-1} k((x_{out})_j, (a_{out})_j) \ \ominus\ (txfee \otimes H)$$

$$= k(\sum_{i=1}^{m}[(x_{in})_i]_\pi, \sum_{i=1}^{m}[(a_{in})_i]_\pi) \ \ominus \ k(\sum_{j=1}^{t-1}(x_{out})_j, \sum_{j=1}^{t-1}(a_{out})_j) \ \ominus \ (txfee \otimes H)$$

(where we invoked the additive homomorphic property of the Pedersen Commitment map k)

$$= [(\sum_{i=1}^{m}[(x_{in})_i]_\pi) \otimes G] \ \oplus \ [(\sum_{i=1}^{m}[(a_{in})_i]_\pi) \otimes H] \ \ominus \ [(\sum_{j=1}^{t-1}(x_{out})_j) \otimes G] \ \ominus \ [(\sum_{j=1}^{t-1}(a_{out})_j) \otimes H] \ \ominus \ [txfee \otimes H]$$

$$=$$

$$\{[\sum_{i=1}^{m}[(x_{in})_i]_\pi \ - \ \sum_{j=1}^{t-1}(x_{out})_j] \otimes G\} \ominus \{[txfee \ + \ \sum_{j=1}^{t-1}(a_{out})_j \ - \ \sum_{i=1}^{m}[(a_{in})_i]_\pi] \otimes H\}$$

$$= \{z \otimes G\} \ \ominus \ \{[txfee \ + \ \sum_{j=1}^{t-1}(a_{out})_j \ - \ \sum_{i=1}^{m}[(a_{in})_i]_\pi] \otimes H\}$$

where $+$ and $-$ are addition and subtraction in modulo l arithmetic over $\mathbb{F}_l$, and where we used $\sum_{i=1}^{m}[(x_{in})_i]_\pi - \sum_{j=1}^{t-1}(x_{out})_j = z \pmod{l}$.

If the transaction amounts don't balance out, then:

$$[txfee \ + \ \sum_{j=1}^{t-1}(a_{out})_j \ - \ \sum_{i=1}^{m}[(a_{in})_i]_\pi] \otimes H \neq 0$$

And since the DL of H in base G is unknown, one can conclude that with overwhelming probability the sender π would not know the DL of $\sum_{i=1}^{m}[(C_{in})_i]_\pi \ \ominus \ \sum_{j=1}^{t}(C_{out})_j$ in base G. The contrapositive statement ensures that if the sender can prove that he knows this DL, then with overwhelming probability, we must have $[txfee \ + \ \sum_{j=1}^{t-1}(a_{out})_j \ - \ \sum_{i=1}^{m}[(a_{in})_i]_\pi] \pmod{l} = 0$.

So by ensuring that:

1. $\sum_{i=1}^{m}[(x_{in})_i]_\pi - \sum_{j=1}^{t-1}(x_{out})_j = z \pmod{l}, \ z \in \mathbb{F}_l^*$ (only known to the sender π), and
2. $\forall i \in \{1, ..., m\}, \ \forall j \in \{1, ...t\}$ the amounts $[(a_{in})_i]_\pi$ and $(a_{out})_j$ remain confined to a pre-defined range $[0, 2^r] \subset \mathbb{F}_l$ (refer to part 8 for more information about the choice of r).

we can follow the same procedure outlined in part 8 and derive the following equivalence:

$$\sum_{i=1}^{m}[(C_{in})_i]_\pi \ \ominus \ \sum_{j=1}^{t}(C_{out})_j = z \otimes G, \ z \in \mathbb{F}_l^*$$

$$\iff \ [\sum_{i=1}^{m}(a_{in})_i]_\pi - \sum_{j=1}^{t}(a_{out})_j = 0$$

The expression $\sum_{i=1}^{m}[(C_{in})_i]_\pi \ \ominus \ \sum_{j=1}^{t}(C_{out})_j = z \otimes G$ showcases z as the private key associated with public key $\sum_{i=1}^{m}[(C_{in})_i]_\pi \ \ominus \ \sum_{j=1}^{t}(C_{out})_j$. No one knows the value of z except the sender (note that this is possible since z is fully defined by the blinding factors $[(x_{in})_i]_\pi$ and $(x_{out})_j$ which are only known to the sender - we will see how so when we allude to relevant portions of the Monero code in the next section). A signature that is verified using public key $\sum_{i=1}^{m}[(C_{in})_i]_\pi \ \ominus \ \sum_{j=1}^{t}(C_{out})_j$ demonstrates that its signer knows the private key z. Consequently, this demonstrates that the

transaction amounts are balanced. That leads us to amend the MLSAG scheme by introducing an additional key to the key vector of each user in the ring. The amended public key matrix becomes:

$$PK = \begin{bmatrix} y_1^1 & .. & y_\pi^1 & .. & y_n^1 \\ .. & .. & .. & .. & .. \\ y_1^m & .. & y_\pi^m & .. & y_n^m \\ \sum_{i=1}^m [(C_{in})_i]_1 \ominus \sum_{j=1}^t (C_{out})_j & .. & \sum_{i=1}^m [(C_{in})_i]_\pi \ominus \sum_{j=1}^t (C_{out})_j & .. & \sum_{i=1}^m [(C_{in})_i]_n \ominus \sum_{j=1}^t (C_{out})_j \end{bmatrix}$$

where $\sum_{i=1}^m [(C_{in})_i]_k,\ k \in \{1, ..., n\}$ denotes the sum of the Pedersen Commitments associated with all the UTXOs of ring member k. **ringCT** consists of conducting an MLSAG signature on the aforementioned public key matrix, where the private-key vector of the sender is given by $[x_\pi^1\ ...\ x_\pi^m\ z]^T$. A valid signature on this public key matrix proves 2 things:

1. That the sender holds the private key vector $[x_\pi^1\ ...\ x_\pi^m]^T$ associated with all m UTXOs used to source the funds (these are the first m rows of the matrix).
2. That the sender knows the secret key z associated with $\sum_{i=1}^m [(C_{in})_i]_\pi \ominus \sum_{j=1}^t (C_{out})_j$ (this is the second row of the matrix).

The resulting scheme hides transaction amounts and safeguards the anonymity of the signer at the same time [9].

4 Monero transactions in practice

Calculation of Pedersen Commitments: In the Monero code base,

- The Pedersen Commitment $(C_{out})_j,\ j \in \{1, ..., t\}$ is known as **outPk**$[j]$**.mask**. The "Pk" suffix probably refers to "Public key" since Pedersen Commitments can be thought of as public keys (recall that they are scalar multiples of G).
- The blinding factor $(x_{out})_j$ associated with $(C_{out})_j,\ j \in \{1, ..., t\}$ is known as **outSk**$[j]$**.mask**. The "Sk" suffix probably refers to "Secret key" since blinding factors are scalars $\in \mathbb{F}_l$.

The calculation of these two values is conducted in the **proveRange** method which can be found in [6]

```
306     //proveRange and verRange
307     //proveRange gives C, and mask such that \sumCi = C
308     //   c.f. http://eprint.iacr.org/2015/1098 section 5.1
309     //   and Ci is a commitment to either 0 or 2^i, i=0,...,63
310     //   thus this proves that "amount" is in [0, 2^64]
311     //   mask is a such that C = aG + bH, and b = amount
312     //verRange verifies that \sum Ci = C and that each Ci is a commitment to 0 or 2^i
313     rangeSig proveRange(key & C, key & mask, const xmr_amount & amount) {
314         sc_0(mask.bytes);
315         identity(C);
316         bits b;
317         d2b(b, amount);
318         rangeSig sig;
319         key64 ai;
320         key64 CiH;
321         int i = 0;
322         for (i = 0; i < ATOMS; i++) {
323             skGen(ai[i]);
324             if (b[i] == 0) {
325                 scalarmultBase(sig.Ci[i], ai[i]);
326             }
327             if (b[i] == 1) {
328                 addKeys1(sig.Ci[i], ai[i], H2[i]);
329             }
330             subKeys(CiH[i], sig.Ci[i], H2[i]);
331             sc_add(mask.bytes, mask.bytes, ai[i].bytes);
332             addKeys(C, C, sig.Ci[i]);
333         }
334         sig.asig = genBorromean(ai, sig.Ci, CiH, b);
335         return sig;
336     }
```

The **proveRange** method takes three arguments:

1. C, which will hold the Pedersen Commitment value associated with a certain output amount. This corresponds to $(C_{out})_j,\ j \in \{1, ..., t\}$ in our earlier notation.

2. *mask*, which will hold the blinding factor value used in the calculation of this Pedersen Commitment. This corresponds to $(x_{out})_j,\ j \in \{1, ..., t\}$ in our earlier notation.

3. *amount*, which is the output amount for which the Pedersen Commitment will be calculated. This amount corresponds to $(a_{out})_j,\ j \in \{1, ..., t\}$ in our earlier notation.

The method operates on *amount* and assigns the corresponding Pedersen Commitment and blinding factor to C and *mask* respectively.

The first step consists in initializing C to the identity element of the group $\{G\}$. This is done by calling the **identity** method found in [5]

```
63      static const key I = { {0x01, 0x00, 0x00,0x00 , 0x00, 0x00, 0x00,0x00 , 0x00, 0x00, 0x00,0x00 , 0x00, 0x00, 0x00,0x00 , 0x00, 0x00, 0x
69      //Creates a zero elliptic curve point
70      inline key identity() { return I; }
71      inline void identity(key &Id) { memcpy(&Id, &I, 32); }
```

The next step iconsists in decomposing *amount* into its binary digits. This is done by calling the **d2b** method found in [7]

```
142     //uint long long to int[64]
143     void d2b(bits  amountb, xmr_amount val) {
144         int i = 0;
145         while (val != 0) {
146             amountb[i] = val & 1;
147             i++;
148             val >>= 1;
149         }
150         while (i < 64) {
151             amountb[i] = 0;
152             i++;
153         }
154     }
```

The method **d2b** takes two arguments, namely *amount* and an array b to store the binary digits. The number of binary digits resulting from this decomposition is capped at a maximum of 64 (i.e., the highest transaction amount allowed is 2^{64} atomic units, where each XMR unit corresponds to 10^{12} atomic units). The upper bound is stored in the variable **ATOM** found in [8].

```
63  //atomic units of moneros
64  #define ATOMS 64
```

For each binary digit $0 \leq i < ATOMS \equiv 64$, the **proveRange** method generates a scalar by calling the **skGen** method found in [4].

```
63      //generates a random scalar which can be used as a secret key or mask
64      void skGen(key &sk) {
65          sk = crypto::rand<key>();
66          sc_reduce32(sk.bytes);
67      }
```

The **skGen** method randomly creates a secret key (i.e., a scalar $\in \mathbb{F}_l$) and assigns it to its argument. As a result, the expression **skGen(**$a_i[i]$**)** (which appears in the **proveRange** method) assigns a random scalar to variable $a_i[i]$. This variable plays the role of the blinding factor associated with binary digit i.

A binary digit can either be equal to 0 or to 1. The method **d2b** stores digit i in variable $b[i]$. If $b[i]$ is 0, then the Pedersen Commitment associated with digit i is set to $C_i[i] \equiv (a_i[i] \otimes G) \oplus (0 \otimes H)$ which is equal to $(a_i[i] \otimes G)$. This calculation is performed by calling the **scalarmultBase** method found in [4]:

```
160     //does a * G where a is a scalar and G is the curve basepoint
161     void scalarmultBase(key &aG,const key &a) {
162         ge_p3 point;
163         sc_reduce32copy(aG.bytes, a.bytes); //do this beforehand!
164         ge_scalarmult_base(&point, aG.bytes);
165         ge_p3_tobytes(aG.bytes, &point);
166     }
```

If $b[i]$ is 1, then the corresponding Pedersen Commitment is set to $C_i[i] \equiv (a_i[i] \otimes G) \oplus (2^i \otimes H)$. Here, $2^i \otimes H$ corresponds to the $H2[i]$ argument fed to the **addKeys1** method invoked in the **proveRange** method. $H2[i]$ is retrieved in the $H2$ table which can be found in [8]. The **addKeys1** method is found in [4].

```
231     //addKeys1
232     //aGB = aG + B where a is a scalar, G is the basepoint, and B is a point
233     void addKeys1(key &aGB, const key &a, const key & B) {
234         key aG = scalarmultBase(a);
235         addKeys(aGB, aG, B);
236     }
```

The blinding factor $mask$ is finally set to $\sum_{i=0}^{ATOMS-1} a_i[i]$. This is done in the **proveRange** method by calling the **sc_add method**.

Lastly, the Pedersen Commitment C is calculated by adding all the $C_i[i]$ computed for $0 \leq i < ATOMS \equiv 64$. To see why this computation yields the desired Pedersen Commitment, note the following:

$$C \equiv (C_{out})_j = ((x_{out})_j \otimes G) \ \oplus \ ((a_{out})_j \otimes H)$$

$$= ((\sum_{i=0}^{ATOMS-1} a_i[i]) \otimes G) \ \oplus \ ((\sum_{i=0}^{ATOMS-1} b[i] \times 2^i) \otimes H)$$

$$= \sum_{i=0}^{ATOMS-1} C_i[i]$$

In the **proveRange** method, this is done by invoking the **addKeys** method found in [4].

```
212     //for curve points: AB = A + B
213     void addKeys(key &AB, const key &A, const key &B) {
214         ge_p3 B2, A2;
215         CHECK_AND_ASSERT_THROW_MES_L1(ge_frombytes_vartime(&B2, B.bytes) == 0, "ge_frombytes_vartime failed at "+boost::lexical_cast<std::s
216         CHECK_AND_ASSERT_THROW_MES_L1(ge_frombytes_vartime(&A2, A.bytes) == 0, "ge_frombytes_vartime failed at "+boost::lexical_cast<std::s
217         ge_cached tmp2;
218         ge_p3_to_cached(&tmp2, &B2);
219         ge_p1p1 tmp3;
220         ge_add(&tmp3, &A2, &tmp2);
221         ge_p1p1_to_p3(&A2, &tmp3);
222         ge_p3_tobytes(AB.bytes, &A2);
223     }
```

The C and $mask$ values are assigned to variables **outPk**$[j]$**.mask** and **outSk**$[j]$**.mask** through the **genRct** method by calling the **proveRange** method. The **genRct** method is found in [6].

```
644     //RingCT protocol
645     //genRct:
646     //   creates an rctSig with all data necessary to verify the rangeProofs and that the signer owns one of the
647     //   columns that are claimed as inputs, and that the sum of inputs  = sum of outputs.
648     //   Also contains masked "amount" and "mask" so the receiver can see how much they received
649     //verRct:
650     //   verifies that all signatures (rangeProogs, MG sig, sum inputs = outputs) are correct
651     //decodeRct: (c.f. http://eprint.iacr.org/2015/1098 section 5.1.1)
652     //   uses the attached ecdh info to find the amounts represented by each output commitment
653     //   must know the destination private key to find the correct amount, else will return a random number
654     //   Note: For txn fees, the last index in the amounts vector should contain that
655     //   Thus the amounts vector will be "one" longer than the destinations vectort
656     rctSig genRct(const key &message, const ctkeyV & inSk, const keyV & destinations, const vector<xmr_amount> & amounts, const ctkeyM &mi
657         CHECK_AND_ASSERT_THROW_MES(amounts.size() == destinations.size() || amounts.size() == destinations.size() + 1, "Different number of
658         CHECK_AND_ASSERT_THROW_MES(amount_keys.size() == destinations.size(), "Different number of amount_keys/destinations");
659         CHECK_AND_ASSERT_THROW_MES(index < mixRing.size(), "Bad index into mixRing");
660         for (size_t n = 0; n < mixRing.size(); ++n) {
661           CHECK_AND_ASSERT_THROW_MES(mixRing[n].size() == inSk.size(), "Bad mixRing size");
662         }
663         CHECK_AND_ASSERT_THROW_MES((kLRki && msout) || (!kLRki && !msout), "Only one of kLRki/msout is present");
664
665         rctSig rv;
666         rv.type = bulletproof ? RCTTypeFullBulletproof : RCTTypeFull;
667         rv.message = message;
668         rv.outPk.resize(destinations.size());
669         if (bulletproof)
670           rv.p.bulletproofs.resize(destinations.size());
671         else
672           rv.p.rangeSigs.resize(destinations.size());
673         rv.ecdhInfo.resize(destinations.size());
675         size_t i = 0;
676         keyV masks(destinations.size()); //sk mask..
677         outSk.resize(destinations.size());
678         for (i = 0; i < destinations.size(); i++) {
679             //add destination to sig
680             rv.outPk[i].dest = copy(destinations[i]);
681             //compute range proof
682             if (bulletproof)
683                 rv.p.bulletproofs[i] = proveRangeBulletproof(rv.outPk[i].mask, outSk[i].mask, amounts[i]);
684             else
685                 rv.p.rangeSigs[i] = proveRange(rv.outPk[i].mask, outSk[i].mask, amounts[i]);
686             #ifdef DBG
687             if (bulletproof)
688                 CHECK_AND_ASSERT_THROW_MES(verBulletproof(rv.p.bulletproofs[i]), "verBulletproof failed on newly created proof");
689             else
690                 CHECK_AND_ASSERT_THROW_MES(verRange(rv.outPk[i].mask, rv.p.rangeSigs[i]), "verRange failed on newly created proof");
691             #endif
692
693             //mask amount and mask
694             rv.ecdhInfo[i].mask = copy(outSk[i].mask);
695             rv.ecdhInfo[i].amount = d2h(amounts[i]);
696             hwdev.ecdhEncode(rv.ecdhInfo[i], amount_keys[i]);
697         }
699         //set txn fee
700         if (amounts.size() > destinations.size())
701         {
702           rv.txnFee = amounts[destinations.size()];
703         }
704         else
705         {
706           rv.txnFee = 0;
707         }
708         key txnFeeKey = scalarmultH(d2h(rv.txnFee));
709
710         rv.mixRing = mixRing;
711         if (msout)
712           msout->c.resize(1);
713         rv.p.MGs.push_back(proveRctMG(get_pre_mlsag_hash(rv, hwdev), rv.mixRing, inSk, outSk, rv.outPk, kLRki, msout ? &msout->c[0] : NULL,
714         return rv;
715     }
```

Among other things, the **genRct** method takes a **destinations** vector as argument. Each element of the vector consists of the address of a relevant recipient of funds for this transaction. The length of the **destinations** vector corresponds to the total number of recipients (eg., in our earlier hypothetical example, it would be 3). The **genRct** method loops through all of them, each time making a call to the **proveRange** method with the following arguments:

1. **outPk**[i]**.mask** which stores the Pedersen Commitment associated with the amount to be sent to the recipient. Note that the "mask" attribute in this case is not a blinding factor. This is a matter of Monero code convention where each key has 2 fields associated with it: 1) A *dest* field, and 2) A *mask* field

    - In case the structure is a secret key (e.g., a Monero amount - recall that amounts are elements of $\mathbb{F}_l$ and hence are scalars, a.k.a. secret keys), the *dest* field would contain the secret key, while the *mask* field would contain the randomly generated blinding factor as described earlier.
    - In case the structure is a public key, *dest* would contain the address and *mask* would contain the Pedersen Commitment of the amount to be transferred to the address. The definitions can be found in [8].

```
90      //containers For CT operations
91      //if it's  representing a private ctkey then "dest" contains the secret key of the address
92      // while "mask" contains a where C = aG + bH is CT pedersen commitment and b is the amount
93      // (store b, the amount, separately
94      //if it's representing a public ctkey, then "dest" = P the address, mask = C the commitment
95      struct ctkey {
96          key dest;
97          key mask; //C here if public
98      };
```

2. **outSk**[i]**.mask** which stores the blinding factor associated with the amount to be transferred to the recipient.

3. **amounts**[i] which corresponds to the amount to be transfered to the recipient.

For each recipient j, this assigns

- The relevant Pedersen Commitment C to **outPk**[j]**.mask**
- The blinding factor or *mask* value to **outSk**[j]**.mask**

Finally, the blinding factor and the amount associated with each recipient are encoded so that they are only known to the sender and to the recipient of the funds. This ensures that the sender is the only entity that knows the value of all the blinding factors associated with UTXO amounts used to source the funds as well as blinding factors associated with the amounts destined to each recipient. In other words, only the sender π would simultaneously know $[(x_{in})_i]_\pi, i \in \{1, ..., m\}$ and $(x_{out})_j, j \in \{1, ..., t-1\}$. As a result, the sender is the only entity that knows $z \equiv \{\sum_{i=1}^{m}[(x_{in})_i]_\pi \ - \ \sum_{j=1}^{t-1}(x_{out})_j\} \pmod{l}$ (introduced in the previous section) that makes ringCT work properly.

The encoding is done through a call to the **ecdhEncode** method in **genRct**. It is found in [4].

```
//Elliptic Curve Diffie Helman: encodes and decodes the amount b and mask a
// where C= aG + bH
void ecdhEncode(ecdhTuple & unmasked, const key & sharedSec) {
    key sharedSec1 = hash_to_scalar(sharedSec);
    key sharedSec2 = hash_to_scalar(sharedSec1);
    //encode
    sc_add(unmasked.mask.bytes, unmasked.mask.bytes, sharedSec1.bytes);
    sc_add(unmasked.amount.bytes, unmasked.amount.bytes, sharedSec2.bytes);
}
```

The **ecdhEncode** method takes 2 arguments:

1. **unmasked**, which has 2 attributes: 1) A blinding factor known as *mask*, and 2) A transaction *amount*.
2. **sharedSec**, which is a a secret key only known to the sender and the recipient of the funds and used to encode the transaction's blinding factor and amount.

The encryption is done as follows:

- The *mask* $(x_{out})_j$ is mapped to $(x_{out})_j + keccak(sharedSec)$, where keccak is the hash function used by Monero.
- The *amount* $(a_{out})_j$ is mapped to $(a_{out})_j + keccak(keccak(sharedSec))$

Building the amended public key matrix: Recall that the amended public key matrix introduced in the previous section was given by:

$$PK = \begin{bmatrix} y_1^1 & .. & y_\pi^1 & .. & y_n^1 \\ .. & .. & .. & .. & .. \\ y_1^m & .. & y_\pi^m & .. & y_n^m \\ \sum_{i=1}^m [(C_{in})_i]_1 \ominus \sum_{j=1}^t (C_{out})_j & .. & \sum_{i=1}^m [(C_{in})_i]_\pi \ominus \sum_{j=1}^t (C_{out})_j & .. & \sum_{i=1}^m [(C_{in})_i]_n \ominus \sum_{j=1}^t (C_{out})_j \end{bmatrix}$$

where $\sum_{i=1}^m [(C_{in})_i]_k$ denotes the sum of the Pedersen Commitments associated with all the UTXOs of ring member $k \in \{1, ..., n\}$.

The calculation of this matrix is performed in the **proveRctMG** method found in [6]. (Note that in the code below, our variable i corresponds to the code's variable j and our variable k corresponds to the code's variable i).

```
//Ring-ct MG sigs
//Prove:
//   c.f. http://eprint.iacr.org/2015/1098 section 4. definition 10.
//   This does the MG sig on the "dest" part of the given key matrix, and
//   the last row is the sum of input commitments from that column - sum output commitments
//   this shows that sum inputs = sum outputs
//Ver:
//   verifies the above sig is created corretly
mgSig proveRctMG(const key &message, const ctkeyM & pubs, const ctkeyV & inSk, const ctkeyV &outSk, const ctkeyV & outPk, const multis
```

```
469         keyM M(cols, tmp);
470         //create the matrix to mg sig
471         for (i = 0; i < cols; i++) {
472             M[i][rows] = identity();
473             for (j = 0; j < rows; j++) {
474                 M[i][j] = pubs[i][j].dest;
475                 addKeys(M[i][rows], M[i][rows], pubs[i][j].mask); //add input commitments in last row
476             }
477         }
483         for (i = 0; i < cols; i++) {
484             for (size_t j = 0; j < outPk.size(); j++) {
485                 subKeys(M[i][rows], M[i][rows], outPk[j].mask); //subtract output Ci's in last row
486             }
487             //subtract txn fee output in last row
488             subKeys(M[i][rows], M[i][rows], txnFeeKey);
489         }
```

For each ring member $k \in \{1, ..., n\}$, the input Pedersen Commitments $[(C_{in})_i]_k,\ i \in \{1, ..., m\}$ are added. Then the output Pedersen Commitments $(C_{out})_j,\ j \in \{1, ..., t\}$ are subtracted.

RCTTypeFull vs. RCTTypeSimple : The signature scheme along with the amended public key matrix that we introduced thus far is known as **RCTTypeFull** (also referred to as Type 1 in Monero's code base). It treats all UTXOs at once as part of a single ring signature structure: if we have m UTXOs and a **mixin count** of $n-1$, **RCTTypefull** creates a public key matrix of size $(m+1) \times n$ and signs the transaction in one go. As we previously noted in our hypothetical example, it is imperative that the index of each UTXO used by the sender be the same (recall that in our hypothetical example the index π was equal to 3 for each of the 2 UTXOs). This is dictated by the structure of the public key matrix.

Monero uses a signature of type **RCTTypeFull** (i.e., of Type 1) when a transaction has only 1 UTXO. Whenever a sender uses more that 1 UTXO to conduct a transfer, Monero invokes a more efficient variant known as RCTTypeSimple (also known as Type 2). An enumeration of Monero's RCT Types is found in [8].

```
220     enum {
221       RCTTypeNull = 0,
222       RCTTypeFull = 1,
223       RCTTypeSimple = 2,
224       RCTTypeFullBulletproof = 3,
225       RCTTypeSimpleBulletproof = 4,
226     };
```

A Type 0 corresponds to a coinbase transaction. Simply put, it is a particular type of transaction issued by a miner whenever a new block is successfully created. It takes no input, but creates new currency units to reward the miner for her successful work. Types 0, 3, and 4 are not within the scope of this work. We now describe the **RCTTypeSimple** variant of the ringCT signature.

We derived the following equality in section 3:

$$\sum_{i=1}^{m}[(C_{in})_i]_\pi \ominus \sum_{j=1}^{t}(C_{out})_j = \{z \otimes G\} \ominus \{[txfee + \sum_{j=1}^{t-1}(a_{out})_j - \sum_{i=1}^{m}[(a_{in})_i]_\pi] \otimes H\}$$

$$\text{where } z \in \mathbb{F}_l^* \text{ is a scalar equal to } [\sum_{i=1}^{m}[(x_{in})_i]_\pi - \sum_{j=1}^{t-1}(x_{out})_j] \pmod{l}$$

Let's define a new set of commitments that we call **pseudo-output commitments** or C_ψ. We create one for each index $i \in \{1, ..., m\}$ as follows:

- $\forall i \in \{1, ..., m-1\}$

  $\{$ Generate random scalar $(x_\psi)_i$

  $\{$ Compute $(C_\psi)_i = [(x_\psi)_i \otimes G] \oplus [[(a_{in})_i]_\pi \otimes H]$

- For $i = m$, set

  $\{$ $(x_\psi)_m = \sum_{j=1}^{t-1}(x_{out})_j - \sum_{i=1}^{m-1}(x_\psi)_i$

  $\{$ $(C_\psi)_m = [(x_\psi)_m \otimes G] \oplus [[(a_{in})_m]_\pi \otimes H]$

The above construction ensures that $\sum_{i=1}^{m}(x_\psi)_i = \sum_{j=1}^{t-1}(x_{out})_j$

We can re-write the original equality as:

$$\{\sum_{i=1}^{m}[(C_{in})_i]_\pi \ominus \sum_{i=1}^{m}(C_\psi)_i\} \oplus \{\sum_{i=1}^{m}(C_\psi)_i \ominus \sum_{j=1}^{t-1}(C_{out})_j \ominus (txfee \otimes H)\}$$

$$= \{z \otimes G\} \oplus \{[\sum_{i=1}^{m}[(a_{in})_i]_\pi - \sum_{j=1}^{t-1}(a_{out})_j - txfee] \otimes H\}$$

Note the following:

① If we can prove that $\sum_{i=1}^{m}[(C_{in})_i]_\pi \ominus \sum_{i=1}^{m}(C_\psi)_i = z \otimes G$, we can conclude that

$$\sum_{i=1}^{m}(C_\psi)_i \ominus \sum_{j=1}^{t-1}(C_{out})_j \ominus (txfee \otimes H) =$$

$$[\sum_{i=1}^{m}[(a_{in})_i]_\pi - \sum_{j=1}^{t-1}(a_{out})_j - txfee] \otimes H$$

② If we can furthermore show that $\sum_{i=1}^{m}(C_\psi)_i \ominus \sum_{j=1}^{t-1}(C_{out})_j \ominus (txfee \otimes H)$ is equal to 0, then we can conclude that $\sum_{i=1}^{m}[(a_{in})_i]_\pi = \sum_{j=1}^{t-1}(a_{out})_j + txfee \pmod{l}$, and hence that the amounts are balanced modulo l

③ If in addition, we can prove that the amounts $[(a_{in})_i]_\pi$ and $(a_{out})_j$ are confined to a pre-defined range $[0, 2^r] \subset \mathbb{F}_l$ (refer to part 8 for more information about the choice of r), then we can conclude that $\sum_{i=1}^{m}[(a_{in})_i]_\pi = \sum_{j=1}^{t-1}(a_{out})_j + txfee$, and hence that the amounts are balanced independent of modulo l arithmetic.

We observe that if $\forall i \in \{1, ..., m\}$, we have $[(C_{in})_i]_\pi \ominus (C_\psi)_i = z_i$ such that $\sum_{i=1}^{m} z_i = z$, then ① will certainly hold. In essence, this corresponds to having a total of m signatures, each signed with a relevant z_i since $[(C_{in})_i]_\pi \ominus (C_\psi)_i$ can be thought of as a public key associated with secret key z_i

So in **RCTTypeSimple**, we do not create a single public key matrix and hence do not apply MLSAG only once. Instead, we create m different public key matrices with each having its own MLSAG. The m public key matrices are elements of $\{G\}^{2\times n}$ and are given by:

$$PK_i = \begin{bmatrix} y_1^i & .. & y_\pi^i & .. & y_n^i \\ [(C_{in})_i]_1 \ominus (C_\psi)_i & .. & [(C_{in})_i]_\pi \ominus (C_\psi)_i & .. & [(C_{in})_i]_n \ominus (C_\psi)_i \end{bmatrix}$$

$[(C_{in})_i]_k$ refers to the Pedersen Commitment associated with the i^{th} UTXO $((i \in \{1, ..., m\})$ of ring member k $(k \in \{1, ..., n\})$. Note that $\forall i \in \{1, ..., m\}$, the value of $(C_\psi)_i$ is the same $\forall k \in \{1, ..., n\}$. We can think of this as being a separate MLSAG on each UTXO used by the sender. It proves 2 things:

1. That the sender holds the private key x_π^i associated with his i^{th} UTXO (this is the first row of the matrix).

2. That the sender knows the secret key z_i associated with $[(C_{in})_i]_\pi \ \ominus \ (C_\psi)_i$ (this is the second row of the matrix).

In Monero's code base, the creation of the pseudo-output commitments $(C_\psi)_k, \ k \in \{1, ..., n\}$ is done in the **genRctSimple** method found in [6]:

```
729     rctSig genRctSimple(const key &message, const ctkeyV & inSk, const keyV & destinations, const vector<xmr_amount> &inamounts, const vecto
730         CHECK_AND_ASSERT_THROW_MES(inamounts.size() > 0, "Empty inamounts");
731         CHECK_AND_ASSERT_THROW_MES(inamounts.size() == inSk.size(), "Different number of inamounts/inSk");
732         CHECK_AND_ASSERT_THROW_MES(outamounts.size() == destinations.size(), "Different number of amounts/destinations");
733         CHECK_AND_ASSERT_THROW_MES(amount_keys.size() == destinations.size(), "Different number of amount_keys/destinations");
734         CHECK_AND_ASSERT_THROW_MES(index.size() == inSk.size(), "Different number of index/inSk");
735         CHECK_AND_ASSERT_THROW_MES(mixRing.size() == inSk.size(), "Different number of mixRing/inSk");
736         for (size_t n = 0; n < mixRing.size(); ++n) {
737           CHECK_AND_ASSERT_THROW_MES(index[n] < mixRing[n].size(), "Bad index into mixRing");
738         }
739         CHECK_AND_ASSERT_THROW_MES((kLRki && msout) || (!kLRki && !msout), "Only one of kLRki/msout is present");
740         if (kLRki && msout) {
741           CHECK_AND_ASSERT_THROW_MES(kLRki->size() == inamounts.size(), "Mismatched kLRki/inamounts sizes");
742         }
743 
744         rctSig rv;
745         rv.type = bulletproof ? RCTTypeSimpleBulletproof : RCTTypeSimple;
746         rv.message = message;
747         rv.outPk.resize(destinations.size());
748         if (bulletproof)
749           rv.p.bulletproofs.resize(destinations.size());
750         else
751           rv.p.rangeSigs.resize(destinations.size());
752         rv.ecdhInfo.resize(destinations.size());
753 
754         size_t i;
755         keyV masks(destinations.size()); //sk mask..
756         outSk.resize(destinations.size());
757         key sumout = zero();
758         for (i = 0; i < destinations.size(); i++) {
759 
760             //add destination to sig
761             rv.outPk[i].dest = copy(destinations[i]);
762             //compute range proof
763             if (bulletproof)
```

```
764                 rv.p.bulletproofs[i] = proveRangeBulletproof(rv.outPk[i].mask, outSk[i].mask, outamounts[i]);
765             else
766                 rv.p.rangeSigs[i] = proveRange(rv.outPk[i].mask, outSk[i].mask, outamounts[i]);
767             #ifdef DBG
768             if (bulletproof)
769                 CHECK_AND_ASSERT_THROW_MES(verBulletproof(rv.p.bulletproofs[i]), "verBulletproof failed on newly created proof");
770             else
771                 CHECK_AND_ASSERT_THROW_MES(verRange(rv.outPk[i].mask, rv.p.rangeSigs[i]), "verRange failed on newly created proof");
772             #endif
773
774             sc_add(sumout.bytes, outSk[i].mask.bytes, sumout.bytes);
775
776             //mask amount and mask
777             rv.ecdhInfo[i].mask = copy(outSk[i].mask);
778             rv.ecdhInfo[i].amount = d2h(outamounts[i]);
779             hwdev.ecdhEncode(rv.ecdhInfo[i], amount_keys[i]);
780         }
781
782         //set txn fee
783         rv.txnFee = txnFee;
784 //        TODO: unused ??
785 //        key txnFeeKey = scalarmultH(d2h(rv.txnFee));
786         rv.mixRing = mixRing;
787         keyV &pseudoOuts = bulletproof ? rv.p.pseudoOuts : rv.pseudoOuts;
788         pseudoOuts.resize(inamounts.size());
789         rv.p.MGs.resize(inamounts.size());
790         key sumpouts = zero(); //sum pseudoOut masks
791         keyV a(inamounts.size());
792         for (i = 0 ; i < inamounts.size() - 1; i++) {
793             skGen(a[i]);
794             sc_add(sumpouts.bytes, a[i].bytes, sumpouts.bytes);
795             genC(pseudoOuts[i], a[i], inamounts[i]);
796         }
797         rv.mixRing = mixRing;
798         sc_sub(a[i].bytes, sumout.bytes, sumpouts.bytes);
799         genC(pseudoOuts[i], a[i], inamounts[i]);
800         DP(pseudoOuts[i]);
801
802         key full_message = get_pre_mlsag_hash(rv,hwdev);
803         if (msout)
804           msout->c.resize(inamounts.size());
805         for (i = 0 ; i < inamounts.size(); i++) {
806             rv.p.MGs[i] = proveRctMGSimple(full_message, rv.mixRing[i], inSk[i], a[i], pseudoOuts[i], kLRki ? &(*kLRki)[i]: NULL, msout ? &m
807         }
808         return rv;
809     }
```

- For each recipient appearing in the **destinations** vector, **genRctSimple** makes a call to the **proveRange** method previouly introduced. It stores the Pedersen Commitment $(C_{out})_j$ associated with output amount $(a_{out})_j$, $j \in \{1, ..., t\}$ in variable **outPk[j].mask**. The corresponding blinding factor $(x_{out})_j$ is stored in variable **outSk[j].mask**.

- **genRctSimple** will then call the **sc_add** method to sum all the the blinding factors **outSk[j]**. The result $(\sum_{j=1}^{t-1}(x_{out})_j)$ is stored in variable **sumout**.

- Each output amount and its corresponding blinding factor are then encoded by calling the **ecdhEncode** method.

- The next step consists in calculating the pseudo-output commitments $(C_\psi)_i$, $i \in \{1, ..., m\}$ and their corresponding blinding factors $(x_\phi)_i$. This is done as follows:

- $\forall i \in \{1, ..., m-1\}$ (where m corresponds to **inamounts.size()**), the blinding factor $(x_\phi)_i$ is randomly generated by calling the method **skGen** and stored in variable $a[i]$.
- The pseudo-output commitment $(C_\psi)_i$ is then calculated by calling the method **genC** on amount $[(a_{in})_i]_\pi$ which is stored in variable **inamounts[i]** and blinding factor $(x_\psi)_i$ stored in variable $a[i]$. $(C_\psi)_i$ is stored in variable **pseudoOuts[i]**.
- The method keeps track of $\sum_{i=1}^{m-1}(x_\psi)_i$ in a variable called **sumpouts**. The sum is calculated by calling the method **sc_add**. Hence **sumpouts** $= \sum_{i=1}^{m-1} a[i]$.
- $(x_\psi)_m$ is then set to $a[m]$ = **sumout** - **sumpouts**. This is done by calling the **sc_sub** method.
- Finally, the pseudo-output commitment $(C_\psi)_m$ is constructed by calling the **genC** method on amount $[(a_{in})_m]_\pi$ (stored in variable **inamounts[m]**) and blinding factor $(x_\psi)_m$ (stored in variable $a[m]$). $(C_\psi)_m$ is stored in variable **pseudoOuts[m]**.

With pseudo-output commitments calculated, **genRctSimple** makes m calls to the **proveRctMGSimple** method found in [6]

```
499     //Ring-ct MG sigs Simple
500     //   Simple version for when we assume only
501     //       post rct inputs
502     //       here pubs is a vector of (P, C) length mixin
503     //   inSk is x, a_in corresponding to signing index
504     //       a_out, Cout is for the output commitment
505     //       index is the signing index..
506     mgSig proveRctMGSimple(const key &message, const ctkeyV & pubs, const ctkey & inSk, const key &a , const key &Cout, const multisig_kLRk
507         mgSig mg;
508         //setup vars
509         size_t rows = 1;
510         size_t cols = pubs.size();
511         CHECK_AND_ASSERT_THROW_MES(cols >= 1, "Empty pubs");
512         CHECK_AND_ASSERT_THROW_MES((kLRki && mscout) || (!kLRki && !mscout), "Only one of kLRki/mscout is present");
513         keyV tmp(rows + 1);
514         keyV sk(rows + 1);
515         size_t i;
516         keyM M(cols, tmp);
517
518         sk[0] = copy(inSk.dest);
519         sc_sub(sk[1].bytes, inSk.mask.bytes, a.bytes);
520         for (i = 0; i < cols; i++) {
521             M[i][0] = pubs[i].dest;
522             subKeys(M[i][1], pubs[i].mask, Cout);
523         }
524         return MLSAG_Gen(message, M, sk, kLRki, mscout, index, rows, hwdev);
525     }
```

The code is self explanatory and the m calls generate m different public key matrices as described earlier. Recall that each matrix is an element of $\{G\}^{2\times n}$.

A validation of the m signatures proves that there exists an element $1 \leq \pi \leq n$ of the ring for which $\sum_{i=1}^{m}[(C_{in})_i]_\pi \ominus \sum_{i=1}^{m}(C_\psi)_i = z \otimes G$ (refer to the observation made about ① earlier).

① then leads us to conclude that

$$\sum_{i=1}^{m}(C_\psi)_i \ominus \sum_{j=1}^{t-1}(C_{out})_j \ominus (txfee \otimes H) = [\sum_{i=1}^{m}[(a_{in})_i]_\pi - \sum_{j=1}^{t-1}(a_{out})_j - txfee] \otimes H$$

The next step is to validate ②, and show that

$$\sum_{i=1}^{m}(C_\psi)_i \ominus \sum_{j=1}^{t-1}(C_{out})_j \ominus (txfee \otimes H) = 0$$

Once proven, it allows us to conclude that $\sum_{i=1}^{m}[(a_{in})_i]_\pi = \sum_{j=1}^{t-1}(a_{out})_j + txfee \pmod{l}$, and hence that the amounts are balanced modulo l. The verification of this step is done as part of the **verRctSimple** method found in [6]. We include below the relevant portion of the method that does the verification.

```
944         if (semantics) {
945           key sumOutpks = identity();
946           for (size_t i = 0; i < rv.outPk.size(); i++) {
947               addKeys(sumOutpks, sumOutpks, rv.outPk[i].mask);
948           }
949           DP(sumOutpks);
950           key txnFeeKey = scalarmultH(d2h(rv.txnFee));
951           addKeys(sumOutpks, txnFeeKey, sumOutpks);
952
953           key sumPseudoOuts = identity();
954           for (size_t i = 0 ; i < pseudoOuts.size() ; i++) {
955               addKeys(sumPseudoOuts, sumPseudoOuts, pseudoOuts[i]);
956           }
957           DP(sumPseudoOuts);
958
959           //check pseudoOuts vs Outs..
960           if (!equalKeys(sumPseudoOuts, sumOutpks)) {
961               LOG_PRINT_L1("Sum check failed");
962               return false;
963           }
```

The variable **sumOutpks** is first initialized to the identity element of the elliptic group. It is then built up iteratively by calling the **addKeys** method. The final result is given by $\sum_{j=1}^{t-1}$ **outPk[j].mask** which is none other than $\sum_{j=1}^{t-1}(C_{out})_j$.

Next, the Pedersen Commitment associated with the miner's $txfee$ and given by $txfee \otimes H$ is added to **sumOutpks**.

A similar procedure is followed to calculate **sumPseudoOuts** $= \sum_{i=1}^{m}$ **pseudoOuts[i]**. This is none other than $= \sum_{i=1}^{m}(C_\psi)_i$.

The 2 sums are subsequently compared and a boolean value returned.

Lastly, the Borromean signature construct (out of the scope of this series) is used to validate ③, and conclude that $\sum_{i=1}^{m}(a_{in})_i = \sum_{j=1}^{t-1}(a_{out})_j + txfee$ (i.e., ensuring that the equality holds independently of modulo l arithmetic).

5 Example of a real Monero transaction

On moneroexplorer.com, we retrieve the transaction with tx hash given by

$$55ca673862c14c7987ef0d5bea2f0d3568da4c946c1d31e6584cb12cae1efafc.$$

Here is a breakdown of the JSON representation of this transaction:


```
JSON representation of tx
Tx hash: : 55ca673862c14c7987ef0d5bea2f0d3568da4c946c1d31e6584cb12cae1efafc
{
 "version": 2,
```


The transaction **version** field is equal to 2. This means that this transaction implements ringCT. This is in contrast to the earlier version 1 which implemented a regular ring signature scheme.


```
"vin": [ {
   "key": {
    "amount": 0,
    "key_offsets": [ 2019406, 1111194, 1398546, 235800, 10617
    ],
    "k_image": "bee6a4ed6a23a841110d434ef7bf010864ccecec4127123b0c9b71051f35a964"
   }
  }, {
   "key": {
    "amount": 0,
    "key_offsets": [ 1414191, 971662, 1571790, 626968, 191640
    ],
    "k_image": "c4c74af87d2293509f16f93dd905edb83608816d702d155c67a1dd0511d89639"
   }
  }
 ],
```


- There are 2 **Vin** sets. This means that 2 UTXOs are used to source funds to transfer to recipients. The sender's UTXOs are concealed in a ring of size 5 each. This means that the **mixin count** is equal to 4.

- The first **Vin** set is identified by the array of **key_offsets** $[2019406, 1111194, 1398546, 235800, 10617]$, while the second is identified by $[1414191, 971662, 1571790, 626968, 191640]$.

- A **key offset** is a relative index corresponding to a particular UTXO. In Monero, all UTXOs holding the same amount value are listed sequentially, and the **key offset** is a way to reference a specific UTXO in that list. The rationale for doing so has to do with the earlier version of Monero. Prior to ringCT, Monero's ring signature scheme had to group UTXOs of the same amount together in order to safeguard the anonymity of the signer. If different amounts were allowed to be grouped together, it would be very likely for the index of the signer to be identified since it would be the only one for which the input/output amount equation balances out. The reasoning is similar to the one we employed earlier when we discussed the shortcoming of using a commitment to 0. The difference is that in the latter case, we operate on Pedersen Commitments, while in the former we operate on the actual amounts. With the advent of ringCT, all UTXO amounts became concealed and given the value 0 as an indication that they are hidden. This is reflected in the **amount** field.

- The **key offsets** associated with the first **Vin** set are then the relative indices of UTXOs with hidden amounts (i.e., whose **amount** field is set to 0). For the first **Vin** set, the first UTXO appears at index 2019406, the second at index $(2019406 + 1111194)$, the third at index $(2019406 + 1111194 + 1398546)$ and so on.

- The **k_image** field holds the key image or tag associated with the signer's UTXO. We will see in part 10 that a UTXO is associated with a **"one-time private key"** and a **"one-time public key"**. This unique pair is used to calculate the key image as described in part 7 of this series. The key image associated with the signer's first UTXO (each ring member has 2 UTXOs in this example) is given by

$$I_{\pi}^{1} = x_{\pi}^{1} \otimes H_2(y_{\pi}^{1})$$

  where the superscript 1 refers to the first set of **Vin** and π refers to the index of the signer in the ring. Recall that the key-image construct ensures that MLSAG is linkable, which in turn helps prevent the doublespending problem.


```
"vout": [ {
  "amount": 0,
  "target": {
   "key": "839e866a6d485da28fa2e2874fda6ad93f26694999597cc91f6c32adba427130"
  }
 }, {
  "amount": 0,
  "target": {
   "key": "04d0f8bbf214059885817b2c454dac6745db5402a324b9dbb6e1f9cb82e8f1b8"
  }
 }
],
```


The above exerpt shows that there are 2 recipients of funds (probably a counterparty and a change address). Here too, the amounts are concealed and the **amount** field is set to 0. Note that the **key** field of each recipient holds a **stealth address** (i.e., unique address) that helps conceal his identity. We will introduce stealth addresses in part 10.


```
"rct_signatures": {
 "type": 2,
 "txnFee": 11478320000,
 "pseudoOuts": [ "d5a06a7a5c75d80ad51617cfb307cdac2e5fb19de72266f653b4122f98916ba6", "9aff5dbd33d3c72475619e0f3e7a9b7d4076aa9482996e7553ee6c33c0569938"],
 "ecdhInfo": [ {
   "mask": "d3773186164a081f5c65df9d6978a3a8c6954134d7206113bcaffb688e799302",
   "amount": "2976df1a00b6915e017e576e341b2ac8013a9206c53196a6743f776443ac6b00"
  }, {
   "mask": "8cdf147d5ca8580155b9b0516533e348d61fbd36d40c57061b83f7b036b30f0c",
   "amount": "811b6a6769684f22b06aadf6f954b7009a3381161a79c9ed3a0a15e9f534090e"
  }],
 "outPk": [ "0bc44debd867d967d87b9514725da730c89ea73c98aa5532ae070d0d56b89fbc", "c017a2e53c7d28d3012b37ff930e24a514693b2cc0680f8850e2fab3ba2c29b3"]
},
```


- The **type** field is set to 2. **Type 1** is for **RCT Type Full**, while **type 2** is for **RCT Type Simple**. Recall that **type 1** is implemented if there is only one UTXO (i.e., **Vin** = 1). If there are more than a single **Vin**, then **type 2** is implemented.

- The **txnfee** field specifies the transaction fee paid to the miners. It is expressed in picoNero or atomic units (recall that each Monero unit corresponds to 10^{12} atomic units).

- The **pseudoOuts** field contains the pseudo-output commitments which correspond to the $(C_{\psi})_i,\ i \in \{1, ..., m\}$ introduced earlier. In this case, $m = 2$,

since there are 2 **Vin** sets.

- The **ecdhInfo** section contains the encoded values of the blinding factor and amount associated with each **Vout**. Recall that

  { **mask** $_j = (x_{out})_j + keccak(sharedSec)$, where keccak is the hash function used by Monero, and sharedSec is a shared secret as was introduced in the previous section.

  { **amount** $_j = (a_{out})_j + keccak(keccak(sharedSec))$

  where $j \in \{1, ...t\}$. In this case $t = 2$, since there are 2 **Vout**'s.

- The **outPk** field corresponds to the output Pedersen Commitments. These are the $(C_{out})_j,\ j \in \{1, ..., t\}$ introduced in the previous section. Here $t = 2$, since there are 2 **Vout**'s.


```
"rangeSigs": [ {
  "asig": "a7f9fccb321ce7a41ef04dd7352187b6be233fad8766cc0633813559e5aad10447a8090635d5d915989d978e91ea53f1ce972e6d9c013c275319d....040e",
  "Ci": "8715a7cbd740c3d3636cb8d8054621f67d60b36141273bf16b0618fabfe90c9a6a1af4d813a591421f1faa0e0c97571e6ad1f55c980082c474f218e8...a266"
}, {
  "asig": "7eabbc6947ffdb4f2263492242ed0026442bb27aa63398454 14f77facfc9650788e395b838e74386ad6cef06dfc408abc7cb30a9745839e638ec2c5...b20c",
  "Ci": "c35d2a63f101ab974b877f39a765070539303a45fded8958c3f75c21096b2c7a22603ee6dc5815dc4ac830301b9a4cc672002f931d0a216cc2ba3842....f9cc"
}],
```


This portion contains information pertaining to the proof that transaction amounts are confined to a specific range (i.e., $[0, 2^r]$ as was described previously). The mechanics of Borromean signatures and range proofs were not included in this work.


```
"MGs": [ {
  "ss": [ [ "d9dd6150b53f8e0b98a0954d67a0d373744f017441b56c4b697a117383758409", "8443e9bf0dee7129963d4902fcba0d3191ea2e1de52bda193975928cc9b1dc08"],
        [ "e7cc7f1531b13322da61907eb45fe3ea2ae1a168a16861be2b86fedd37d0e708", "e5d9ce33d9a87e2a3c9f6a5360123567b028f373c20b0d368958f98ec588210d"],
        [ "cb1f33d681f32feff3c9c630dff25c652ba2dfd354926664de11433f69f7100d", "39af4123a4ad84b4be1123dd76229a0e765bf459ef9a6a47c1030c19c06e6c09"],
        [ "444f6107c7efd4cc296e21d67635515d24aadb7cc0d2cbecf948234bbb0df10e", "9f8d0116243b811d46ccab91d7d3119da0cab18ca23649027e775304a1b64a07"],
        [ "fa17f0ceb131d86a48905bffee03a5eeb9ce0a057536a7aba0e080b815279802", "c35e121c93875219e680fb7f9d103ee3baf0b0a153a2d70884612afb40271c0c"]],

  "cc": "35d33eb0b6489412ff08a8bc2ed5aeeb2519d5c1f4077bcc1e4456b22a623d0c"
}, {

  "ss": [ [ "01e4d0e7b7325fb13dc5bbe2fb9408ce127b1af2619d80681f22f3dc8bc6e601", "82d02e2b93a72e28a4daf1471f9f87a60df9b787f4abe6ddc75ca6cdb3c53700"],
        [ "2af63498aef39909f0735d1e66b23f49800cd502a3b7e0a3b41f673ca5413507", "1bb676236f565bea4ff9c9c49b80f553643808182ae83251a4ddf97664c0a04"],
        [ "db2d33b861d7481fe0f22c19a9f62eb0a7109d94a53d86dd0cf997e943706c0f", "eca8992e274f1ea7b0594484cfc52793dd6f46d262d2acdbc2159a97c664c80e"],
        [ "8784c332f70332605535a0b8dac55b67697aa1f199621b0fe4eab66a87aad603", "c27c737a4031316b0dbabf4f507612ef259b74326994f7d333dab294abb1eb02"],
        [ "4c09932ebc71ba294c8c48097b1ea0580bf093337ca6eecebf3d48b0029c7f0a", "b8920246e423a2dc4af6f6a5f72091ba2267c2b97fa749a34498b2031e3b5504"]],

  "cc": "ad50197c3c4826207c7812b4f27ee2a7cff5caf431364c8adcd2c72ef505b107"
}]
```


Since this transaction implements RCTTypeSimple, it creates 2 amended public key matrices PK_1 and PK_2 (1 for each **Vin** set). It then runs an MLSAG on each matrix. As we previously saw in part 7 of this series, an MLSAG signature issued by signer π on message m and public key matrix PK_i is of the form:

$$\sigma_\pi(m, PK_i) = (I_\pi^1, ..., I_\pi^m, c_1, r_1^1, ..., r_1^m, ..., r_n^1, ..., r_n^m)$$

In this case $m = 2$ (since the matrix has 2 rows) and $n = 5$ (since the **mixin count** is equal to 4). Each of the 2 signatures will then be of the form:

$$\sigma_\pi(m, PK_i) = (I^1_\pi, I^2_\pi, c_1, r^1_1, r^2_1, ..., r^1_5, r^2_5)$$

- The **ss** values correspond to the r^j_i's where for example in the first MLSAG signature, ["$d9dd...8409$","$8443...b1dc$"] corresponds to $[r^1_1, r^2_1]$.
- The **cc** value corresponds to c_1 that appears in the MLSAG signature

References

[1] B. Bunz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short proofs for confidential transactions and more. *Stanford*, 2016.

[2] knaccc. What is the transaction id. https://monero.stackexchange.com/questions/5660/what-is-the-transaction-id-and-how-its-calculated.

[3] G. Maxwell and A. Poelstra. Borromean ring signatures. -, 2015.

[4] Monero. rctops.cpp. https://github.com/monero-project/monero/blob/master/src/ringct/rctOps.cpp.

[5] Monero. rctops.h. https://github.com/monero-project/monero/blob/master/src/ringct/rctOps.h.

[6] Monero. rctsigs.cpp. https://github.com/monero-project/monero/blob/master/src/ringct/rctSigs.cpp.

[7] Monero. rcttypes.cpp. https://github.com/monero-project/monero/blob/master/src/ringct/rctTypes.cpp.

[8] Monero. rcttypes.h. https://github.com/monero-project/monero/blob/master/src/ringct/rctTypes.h.

[9] S. Noether and A. Mackenzie. Ring confidential transactions. *Monero Research Lab*, 2016.