FCMP++

Luke "Kayaba" Parker

April 27, 2024

Abstract

FCMP++, shortened from FCMP+SA+L, short for Full-Chain Membership Proofs + Spend Authorization + Linkability, are an accomplishment of full-set privacy over the existing RingCT protocol used within Monero. This document serves as a specification and implementation reference.

1 Background

Monero is currently planning to deploy the Seraphis upgrade, most notable for its definition as a composition which distinguishes membership from spend authorization. With that, we are allowed much more efficient membership proofs (enabling full-set privacy). RingCT, which fails to so distinguish, was considered infeasible for full-set privacy for that reason and due to how its linking tags are defined.

With Seraphis, outputs are defined as $k_0 \cdot G_0 + k_1 \cdot G_1 + k_2 \cdot G_2$ with the linking tag defined as $(k_2/k_1) \cdot J$. This allows k_0 to be re-randomized without malleating the linking tag.

With a new definition of linking tags, it forces creation of a new anonymity set (with explicit migration) to prevent double-spending of already spent outputs (which would now be given a new linking tag if 'imported' by simply adding a $1G_2$ term).

With RingCT, outputs are defined as $O = x \cdot G$ with the linking tag defined as $x \cdot \mathsf{HashToPoint}(O)$. To re-randomize x would be to malleate the linking tag.

2 New Output Key Definition

Since we cannot re-define the linking tags, we re-define the output key. We introduce a new generator, T, re-define output keys as $O = x \cdot G + y \cdot T$ (where $y = 0$ for all existing outputs), and preserve the linking tag definition of $x \cdot \mathsf{HashToPoint}(O)$. This causes all existing outputs to maintain their linking tags while giving us a term to re-randomize.

3 Proof Separation

Seraphis defines four distinct proofs.

1. Membership Proof.

   The tuple of the commitment being spent by this input and the output key associated with it is a member of the set of outputs within this blockchain.

2. "Ownership and Unspentness" (Spend Authorization and Linkability).

   The transaction created is authorized by the possessor of the private keys for the output key. The commitment spent has not been prior spent.

3. Balance Proof.

   This transaction does not create outputs worth more than the inputs spent.

4. Range Proof.

   The outputs do not overflow the order of the elliptic curve in order to cheat the balance proof.

We appreciate and maintain these four definitions. We focus on the first two, currently jointly satisfied by CLSAG. We define one proof for the former and compose it with a new proof for the latter.

3.1 New Membership Proof

$\mathsf{OutputSet}$ is defined as a set of (O, I, C) tuples (where O is the output key, $I = \mathsf{HashToPoint}(O)$, and C the amount commitment).

For an output tuple (O, I, C), the prover samples scalars r_o, r_i, r_j, r_c. They then prove knowledge of those scalars and:

$\tilde{O} = O + r_o \cdot T$

$\tilde{I} = I + r_i \cdot U$

$R = r_i \cdot V + r_j \cdot T$

$\tilde{C} = C + r_c \cdot G$

$(O, I, C) \in \mathsf{OutputSet}$

while only publishing $\tilde{O}, \tilde{I}, \tilde{C}, R$ (the 'input tuple').

This does not require knowledge of x, y (nor the opening of the Pedersen commitment C) and accordingly can be produced by any party trusted with sender privacy. Exactly how this membership proof occurs is left as a black box to the rest of these designs, yet further specified later.

3.2 New Spend Authorization and Linkability Proof

For an input tuple $\tilde{O}, \tilde{I}, \tilde{C}, R$, and linking tag L, prove knowledge of x, y', r_i such that $\tilde{O} = x \cdot G + y' \cdot T, R = r_i \cdot V + r_j \cdot T, \tilde{I} = I + r_i \cdot U$, and $L = x \cdot I$ (while binding to the transaction hash). To be explicit, $y' = y + r_o$.

We instantiate this literally with a Bulletproof+ (https://eprint.iacr.org/2020/735) composed with a Generalized Schnorr Protocol (https://eprint.iacr.org/2009/050).

We start by inlining the math for the Bulletproof+ Weighted-Inner Product proof (in its single round form with its $y = 1$, as that's all we need).

1. Prover samples $r_p, \alpha, \beta, \mu, \delta$ and publishes:
   - $P = x \cdot G + r_i \cdot V + (x * r_i) \cdot U + r_p \cdot T$
     This is the P from the statement of Bulletproof+'s weighted-inner-product proof for $\boldsymbol{g} = [G], \boldsymbol{h} = [V], g = U, h = T$ (in its own notation).
   - $A = \alpha \cdot G + \beta \cdot V + ((\alpha * y) + (\beta * x)) \cdot U + \delta \cdot T$
   - $B = (\alpha * \beta) \cdot U + \mu \cdot T$
2. Verifier sends challenges e.
3. Prover publishes:
   - $s_\alpha = \alpha + e \cdot x$
   - $s_\beta = \beta + e \cdot r_i$
   - $s_\delta = \mu + \delta * e + r_p * (e * e)$
4. Verifier checks:
   - $(e * e) \cdot P + e \cdot A + B = (s_\alpha * e) \cdot G + (s_\beta * e) \cdot V + (s_\alpha * s_\beta) \cdot U + s_\delta \cdot T$

We now have $P = x \cdot G + r_i \cdot V + (x * r_i) \cdot U + r_p \cdot T$ for some x, r_i, r_p. We set $P' = P - \tilde{O} - R$. For $P, \tilde{O}$, the only G term is for x. If the xs are consistent, P' will not have a G term. For P, R, the only V term is for r_i. If the r_is are consistent, P' will not have a V term. For an honest prover, this leaves $P' = (x * r_i) \cdot U + (r_p - y' - r_j) \cdot T$.

We now compose this with a Generalized Schnorr Protocol to:

- Open P' over U, T, proving x and r_i were consistent.
- Prove the linking tag is well-formed.

1. Prover samples r_x, r_y, r_z, r_{r_p} and publishes:
   - $R_O = r_x \cdot G + r_y \cdot T$
     This is a commitment to the nonces we use when showing we know an opening of $\tilde{O}$.

- $R_P = r_z \cdot U + r_{r_p} \cdot T$
  This is a commitment to the nonces we use when showing we know an opening of P'.
- $R_L = r_x \cdot \tilde{I} + r_z \cdot -U$
  This is a commitment to the nonces we use when showing we know a consistent opening of L.

2. Verifier sends challenges c.
3. Prover publishes:
   - $s_x = r_x + c \cdot x$
   - $s_y = r_y + c \cdot y'$
   - $s_z = r_z + c \cdot (x * r_i)$
     This implicitly makes z, never explicitly defined, $x * r_i$.
   - $s_{r_p} = r_{r_p} + c \cdot (r_p - y' - r_j)$
     $r_p - y' - r_j$ is the coefficient for the T term of P'.
4. Verifier checks:
   - $R_O + c \cdot \tilde{O} == s_x \cdot G + s_y \cdot T$
     This verifies the opening of $\tilde{O}$.
   - $R_P + c \cdot P' == s_z \cdot U + s_{r_p} \cdot T$
     This verifies the opening of P' over U, T.
   - $R_L + c \cdot L == s_x \cdot \tilde{I} + s_z \cdot -U$
     This verifies the opening of L as $x \cdot \tilde{I} + (x * r_i) \cdot -U$. Since $x \cdot \tilde{I}$ is $x \cdot I + (x * r_i) \cdot U$, this will remove the additional term (proved correct and consistent with the Bulletproof+), leaving $x \cdot I$ (the intended linking tag).

In the future, we should review specialized proofs. It would not be surprising if s_x can be replaced for just s_α, along with a few other similar tweaks.

4 The Literal Membership Proof

With the proofs separated, and spend authorization satisfied, we still need membership. We prove membership over the entire output set using an arithmetic circuit verifying a Merkle tree proof. Since Merkle tree proofs have O(log n) time complexity, a proof with O(n) complexity still achieves a O(log n) solution to the problem statement.

This makes the requirement a sufficiently performant, zero-knowledge proof for the satisfaction of an arithmetic circuit, yet not explicitly a SNARK (which would mandate sublinear time complexity).

4.1 Arithmetic Circuit Proof

Bulletproofs are a trustless proof already used within Monero, which can be used to prove arithmetic circuit proofs are satisfied. We solely need a hash function for the Merkle tree, the most efficient being a Pedersen hash.

A Pedersen hash is a collision-resistant hash where for each word hashed, an additional term is added to a Pedersen Vector Commitment. Finding a collision implies finding the discrete log relationship of the generators used.

Creating Pedersen hashes in-circuit would be too expensive. Bulletproofs offer arithmetic over the scalar field of the curve, yet Pedersen hashes produce an output over the field the curve is defined over. Accordingly, it'd require non-native arithmetic. If we instead use an elliptic curve whose scalar field is the field of the curve we're performing the hash with (an 'embedded' or 'towered' curve), we'd have a cost of 256 in-circuit multiplications per word hashed (for a 256-bit elliptic curve).

We instead use Generalized Bulletproofs, a proposed modification to Bulletproofs currently undergoing work to obtain security proofs. Generalized Bulletproofs can be used to output Pedersen hashes at effectively no additional cost, although it cannot operate over its hashes (and we do need to operate over a series of hashes, due to discussing a Merkle tree). We solve this by operating over output hashes with a new Generalized Bulletproof, this one on a curve whose scalar field is the prior curve's field. This continues for each layer of the Merkle tree, and is most efficiently realizable with a curve cycle (a curve whose scalar field is the field of a curve whose scalar field is the field for the original curve). With a curve cycle, one can alternate between the two curves (and two proofs) without needing additional curves/proofs per layer.

The ability to produce Pedersen hashes also enables forming challenges off partial transcripts of the witness. By defining a Pedersen hash of the first n variables within the circuit, we can hash that outside of the circuit with a traditional hash function to obtain a challenge usable for the rest of the circuit. This allows interactive proofs within the circuit (due to the Fiat-Shamir transform), and with it, more efficient gadgets.

For the concept of Generalized Bulletproofs, and opening a Merkle tree using Pedersen hashes with a pair of Bulletproofs over a curve cycle in such an efficient manner, please thank the authors of Curve Trees, https://eprint.iacr.org/2022/756.

This proposal applies their work to RingCT and performs further optimizations via the usage of divisors, yet does not claim to be novel.

4.1.1 Alternatives

Bulletproofs and Bulletproofs+ both define commitments to the witness (A_I, A respectively). These commitments are Pedersen hashes of the entire witness. If these commitments are moved into the statement, we can independently use Bulletproofs+ zk-WIP argument (or ideally a tailored version of it, as we don't need the inner-product functionality) to prove subsets of the commitment are validly formed. By defining the subsets as the variables we want hashed for the Merkle tree, we achieve Pedersen hashing with non-negligible overhead (roughly 3x in verification time, with bandwidth scaling near-linear to the amount of hashes performed).

Distinctly, we could discuss modifying the Bulletproof(+) statement to explicitly support such subsets, presumably via an additional challenge. This would have minimal overhead (an amount comparable to Generalized Bulletproofs if accomplished), be less expressive than Generalized Bulletproofs, yet largely as performant in both bandwidth and verification time.

Neither of these ideas are put forth as guaranteed fallbacks at this time. Solely likely sufficiently performant alternatives if there is an issue with proving Generalized Bulletproofs secure which we can discuss researching.

4.2 Towering Curve Cycle

Since we're performing full-chain membership proofs over Monero's RingCT, and Monero's RingCT uses Ed25519, we need a curve cycle involving Ed25519. Unfortunately, a curve cycle with Ed25519 is impossible for non-trivial curves. This is due to Ed25519 having a cofactor of 8. Instead of forming a curve cycle with Ed25519, we find a prime-order curve whose scalar field is Ed25519's field, and then form a curve cycle with that. This is referred to as a towering curve cycle, due to it being a curve cycle where one curve towers (stands over, not forming a cycle with) Ed25519.

tevador proposed Helios and Selene for an efficient towering curve cycle, documenting the deterministic method used to find them and their security. Please see https://gist.github.com/tevador/4524c2092178df08996487d4e272b096.

4.3 Gadgets

We define an arithmetic circuit with two parts to its statement.

- $a_l * a_r = a_o$

  For vectors a_l, a_r, a_o, a_o is the vector multiplication of a_l, a_r.

- $W_l * a_l + W_r * a_r + W_o * a_o + W_v * V = c$

  W_l, W_r, W_o, W_v, c have an equal amount of columns. An instance of this formula, referred to as a constraint, exists per column. Within an instance, we refer to W_l, W_r, W_o, W_v, c as the column relevant (and not the overall structure).

  W_l, W_r, W_o, W_V has as many rows as a_l, a_r, a_o. W_v additionally has a depth equivalent to the amount of Pedersen vector commitments. c is a single value.

  $W_{l,r,o} * a_{l,r,o}$ is $\sum_{i=0}^{a_l.\mathsf{len}()} W_{l,r,o_i} * a_{l,r,o_i}$ (where l, r, o refers to one of the relevant three).

  $W_v * V$ is $\sum_{i=0}^{V.\mathsf{len}()} \sum_{j=0}^{a_l.\mathsf{len}()} W_{v_{i,j}} * V_{i,j}$

  Please note W_v, V is defined by Bulletproofs(+) to be the Pedersen commitments accessible within the circuit and associated weights. We define V as the Pedersen vector commitments, with the necessary sub-indexing added. While Generalized Bulletproofs still supports Pedersen commitments, we drop the functionality from consideration and don't bother notating it due to lack of use.

A gadget is a series of a_l, a_r (and respective a_o) rows and columns in W_l, W_r, W_o, W_v, c. They are intended to be reusable snippets we can independently formally verify as correct/create security proofs for.

We define $a_l.\mathsf{push}$, $a_r.\mathsf{push}$, $V_i\emptyset$ as functions pushing a value onto the respective vector, returning a constrainable reference. Once $a_l.\mathsf{push}$, $a_r.\mathsf{push}$ have been called, the push to a_o is implicit with a constrainable reference retrievable with $\mathsf{a_o.last()}$.

4.3.1 Equality

```
$equality(a, b) \rightarrow$
    $1\ a + -1\ b = 0$
```

By adding b to both sides, we get $a = b$.

4.3.2 Inverse

```
$\mathsf{Inverse}(a) : z \rightarrow$
    $a_2 = a_l.\mathsf{push}(a)$
    $z = a_r.\mathsf{push}(a^{-1})$
    $\mathsf{Equality}(a, a_2)$
    $1\mathsf{a_o}.\mathsf{last}() = 1$
    return $z$
```

For $a * z = 1$, we can rewrite this as $z = 1/a$ (which is the multiplicative inverse of a).

This also serves to prove $a \neq 0$ since if it was 0, it would produce a product of 0 (which is not 1).

4.3.3 Inequality

```
$inequality(a, b) \rightarrow$
    $c = a - b$
    $a_l.\mathsf{push}(c)$
    $a_r.\mathsf{push}(c^{-1})$
    $1\ a + -1\ b + -1\ c = 0$
    $1\ \mathsf{a_o}.\mathsf{last}() = 1$
```

We constrain c to be $a - b$. Then we constrain c to having a multiplicative inverse. If $a == b$, c will be 0, and $\mathsf{a_o}.\mathsf{last}()$ must be 0 (since any number multiplied by 0 is 0).

4.3.4 Bit

```
$\mathsf{Bit}(b) \rightarrow$
    $b_2 = a_l.\mathsf{push}(b)$
    $b_{sub} = a_r.\mathsf{push}(b - 1)$
    $\mathsf{Equality}(b, b_2)$
    $1\ b + -1\ b_{sub} = 1$
    $1\ \mathsf{a_o}.\mathsf{last}() = 0$
```

Over a prime field, the multiplication of any two non-zero values will be a non-zero value. Accordingly, one of the values has to be zero for the second constraint to hold true. If the left value, b, is zero, b is a valid bit. If the right value is zero, since the right value is constrained to being the left value minus one, the left value must be one (a valid bit).

4.3.5 Member of List

```
$\mathsf{MemberOfList}(L, m) \rightarrow$
    $l = a_l.\mathsf{push}(L_0 - m)$
    $1\ L_0 + -1\ m + -1\ l = 0$
    $\forall\ 1 <= i < L.\mathsf{length}() - 1:$
      $r = a_r.\mathsf{push}(L_i - m)$
      $1\ L_i + -1\ m + -1\ r = 0$
      $o = \mathsf{a_o}.\mathsf{last}()$
      $l = a_l.\mathsf{push}(o))$
      $1\ o + -1\ l = 0$
    $r = a_r.\mathsf{push}(L.\mathsf{last}() - m)$
    $1\ L.\mathsf{last}() + -1\ m + -1\ r = 0$
    $1\ \mathsf{a_o}.\mathsf{last}() = 0$
```

This defines a carried multiplication of $L_i - m$, which will be 0 if and only if $L_i = m$. The multiplication of two values will be 0 if and only if one factor is 0 (since we're operating over a prime field). Accordingly, the carried multiplication will be 0 only if at least one $L_i = m$.

4.3.6 On Curve

```
$\mathsf{OnCurve}_{\mathsf{a,b,c}}(x, y) \rightarrow$
    $y_2 = a_l.\mathsf{push}(y)$
    $y_3 = a_r.\mathsf{push}(y)$
    $y^2 = \mathsf{a_o}.\mathsf{last}()$
    $\mathsf{Equality}(y, y_2)$
    $\mathsf{Equality}(y, y_3)$
    $x_2 = a_l.\mathsf{push}(x)$
    $x_3 = a_r.\mathsf{push}(x)$
    $x^2 = \mathsf{a_o}.\mathsf{last}()$
    $\mathsf{Equality}(x, x_2)$
    $\mathsf{Equality}(x, x_3)$
    $x_2^2 = a_l.\mathsf{push}(x^2)$
    $x_4 = a_r.\mathsf{push}(x)$
    $x^3 = \mathsf{a_o}.\mathsf{last}()$
    $\mathsf{Equality}(x^2, x_2^2)$
    $\mathsf{Equality}(x, x_4)$
    $-1\ y^2 + 1\ x^3 + a\ x^2 + b\ x + 1\ c$
```

This evaluates the curve formula of the embedded curve (written in the form $y^2 = x^3 + a\ x^2 + b\ x + c$).

4.3.7 Incomplete Addition

```
$\mathsf{IncompleteAddition}(x_0, y_0, x_1, y_1, x_2, y_2) \rightarrow$
    $\delta = (y_1 - y_0)/(x_1 - x_0)$
    $\delta = a_l.\mathsf{push}(\delta)$
    $r = a_r.\mathsf{push}(x_1 - x_0)$
    $1\ x_1 + -1\ x_0 + -1\ r = 0$
    $1\ y_1 + -1\ y_0 + -1\ \mathsf{a_o}.\mathsf{last}() = 0$
    $\delta_1 = a_l.\mathsf{push}(\delta)$
    $r = a_r.\mathsf{push}(x_2 - x_0)$
    $\mathsf{Equality}(\delta, \delta_1)$
    $1\ x_2 + -1\ x_0 + -1\ r = 0$
    $-1\ y_2 + -1\ y_0 + -1\ r = 0$
    $\delta_2 = a_l.\mathsf{push}(\delta)$
    $\delta_3 = a_r.\mathsf{push}(\delta)$
    $\mathsf{Equality}(\delta, \delta_2)$
    $\mathsf{Equality}(\delta, \delta_3)$
    $1\ x_0 + 1\ x_1 + 1\ x_2 + -1\ \mathsf{a_o}.\mathsf{last}() = 0$
```

This has a precondition $x_0 \neq x_1$.

4.3.8 Permissible for Hashing

```
$\mathsf{Permissible}_{\mathsf{a,b}}(y) \rightarrow$
    $p = (a * y) + b$
    $l = a_l.\mathsf{push}(sqrt(p))$
    $r = a_r.\mathsf{push}(sqrt(p))$
    $1\ l + -1\ r = 0$
    $1\ \mathsf{a_o}.\mathsf{last}() - a\ y = b$
```

In order to not have have to hash points as their x and y coordinates (which would require a tuple set membership invocation and increase the cost of calculating the Pedersen hash), we define an arithmetic hash ($(a * y) + b$) and use it such that for any x coordinate, only one y coordinate will be accepted. This lets us drop the y coordinate without concern of the prover providing the negative y coordinate (negating the discrete logarithms, enabling double spends).

We consider a y coordinate permissible if $\mathsf{permissible}(y)$ and $!\mathsf{permissible}(-y)$. This can be checked in-circuit with just one multiplication, making it incredibly efficient.

This cannot be used with linking tag generators, yet can be used for all other points we hash.

Full credit to this idea goes to the authors of Curve Trees, https://eprint.iacr.org/2022/756.

4.4 Interactive Gadgets

The rest of the gadgets we define are interactive. This bounds all of their arguments to being within Pedersen vector commitments, and has them called with an additional, per-gadget series of challenges chl. chl is derived from hashing all of the Pedersen vector commitments with per-gadget further derivation.

4.4.1 Tuple Member of Set

```
$\mathsf{TupleMemberOfList}(L, m) \rightarrow$
    $\forall\ 0 <= i < L.\mathsf{length}()$ :
      $\forall\ 0 <= j <$:
        $L_i = \sum_{j=0}^{L_0.\mathsf{length}()} chl_j * L_{i,j}$
    $m = \sum_{j=0}^{m.\mathsf{length}()} chl_j * m_j$
    $\mathsf{MemberOfList}(L, m)$
```

Please note the notation here is an evolution of prior syntax. Arguments are generally considered a constrainable reference (a_{l_0}), yet here it's a linear combination (such as 1 a_{l_0} + 2 a_{r_0}), ignoring the bound on being present in a Pedersen vector commitment). Technically, there's no distinction (since a_{l_0} expands to 1 a_{l_0}), yet it's important to acknowledge.

For this to have an attack in polylogarithmic time, it'd presumably need to be converted to the birthday problem. Since changing the tuple claimed to be a member (the left-hand side) changes the effective list (each member being a potential right-hand side), the two sides aren't independent and Wagner's algorithm isn't usable to find an efficient solution.

4.4.2 Discrete Log Proof

We use elliptic curve divisors, as posited by Liam Eagen (https://eprint.iacr.org/2022/596).

```
$\mathsf{DivisorChallenge}(d_y, d_{yx_{0..m}}, d_{x_{1..n}}, d_0, \delta, \mathsf{chl}) \rightarrow$
  $p_{0_{n_0}} = (3 * x^2 + A)/(2 * y)$
  $p_{0_{n_1}} = \sum_{j=0}^{m} (p_{0_{n_w}} * x^{j+1}) * d_{yx_j} + (p_{0_{n_w}} * d_y)$
  $p_{0_{n_2}} = y * d_{yx_0} + \sum_{j=1}^{m} ((j+1) * y * x^j) * d_{yx_j} + \sum_{i=1}^{n} ((i+1) * x^i) * d_{x_i} + 1$
  $p_{0_n} = p_{0_{n_1}} + p_{0_{n_2}}$
  $p_{0_d} = \mathsf{chl.y} * d_y + \sum_{i=0}^{m} (\mathsf{chl.y} * \mathsf{chl.x}^{(i+1)}) * d_{xy_i} + \sum_{i=1}^{n} \mathsf{chl.x}^{(i+1)} * d_{x_i} + 1 * \mathsf{chl.x} + 1 * d_0$
  $p_{0_d} = \mathsf{Inverse}(p_{0_d})$
  $\mathsf{Equality}(a_l.\mathsf{push}(p_{0_n}), p_{0_n})$
  $\mathsf{Equality}(a_r.\mathsf{push}(p_{0_d}), p_{0_d})$
  $p_0 = \mathsf{a_o.last}()$
  $p_{1_n} = 2 * \mathsf{chl.y}$
  $p_{1_d} = (-\delta \cdot p_{1_n}) + 3 * \mathsf{chl.x} + A$
  $p_{1_d} = \mathsf{Inverse}(p_{1_d})$
  $\mathsf{Equality}(a_l.\mathsf{push}(p_{1_n}), p_{1_n})$
  $\mathsf{Equality}(a_r.\mathsf{push}(p_{1_d}), p_{1_d})$
  $p_1 = \mathsf{a_o.last}()$
  $\mathsf{Equality}(a_l.\mathsf{push}(p_0), p_0)$
  $\mathsf{Equality}(a_r.\mathsf{push}(p_1), p_1)$
  $\mathsf{return}\ \mathsf{a_o.last}()$
```

d_y, d_{yx}, d_x, d_0 are the coefficients for the divisor (zero-indexed, such that d_{x_0} is the coefficient for x^1). The divisor is enforced to be non-zero by using 1 for the coefficient for x^1.

A is the A from the curve equation $y^2 = x^3 + A * x + B$.

Please note we define linear combinations $p_{0_n}, p_{0_d}, p_0, p_{1_n}, p_{1_d}, p_1$, push them as if they were values (which they can be to the prover), and then constrain off them as linear combinations.

```
$\mathsf{DiscreteLog_G}(d_y, d_{yx_{0..m}}, d_{x_{1..n}}, d_0, s_{0..255}, x, y) \rightarrow$
  $\forall\ 0 <= i < 255 :$
    $\mathsf{Bit}(s_i)$
  $\mathsf{chl}_0 = \mathcal{H}_{point}(\mathsf{chl}_0)$
  $\mathsf{chl}_1 = \mathcal{H}_{point}(\mathsf{chl}_1)$
  $\mathsf{chl}_2 = -(\mathsf{chl}_0 + \mathsf{chl}_1)$
  $\delta = (\mathsf{chl}_1.\mathsf{y} - \mathsf{chl}_0.\mathsf{y}) * (\mathsf{chl}_1.\mathsf{x} - \mathsf{chl}_0.\mathsf{x})^{-1}$
  $\mu = \mathsf{chl}_1.y - (\delta * \mathsf{chl}_1.x)$
  $\sum_{i=0}^{2} \mathsf{DivisorChallenge}(d_y, d_{yx}, d_x, d_0, \delta, \mathsf{chl}_i) - \sum_{i=0}^{254} (\mu - (G_i.\mathsf{y} - (\delta * G_i.\mathsf{x})))^{-1} * s_i = 0$
```

For a generator G, both the prover and the verifier calculate $G_0 = 2^0 \cdot G, ..., G_{254} = 2^{254} \cdot G$. For a scalar for the embedded curve s, the prover creates a 255-bit decomposition $b_0, ..., b_{254}$. The prover creates a Pedersen vector commitment for the bits and the divisor interpolating $s \cdot G$ with the relevant powers of G.

We then perform a challenged evaluation of the divisor, and for the right-hand side compared to (the sum of each interpolated point also so challenged), we only include terms selected by the bits.

While evaluating divisors (a polynomial), we consider the evaluation not as $a * x^2 + b * x + c$, yet $x^2 * a + x * b + c$. The latter lines up with the constraint system's definition (since our x is public). For each C_i,

Please note if this discrete log is reused in other proofs, it's unnecessary to again prove it's a valid bitstring.

4.4.3 Discrete Log Proof of Knowledge

$\mathsf{DiscreteLogProofOfKnowledge_G}(d_y, d_{yx_{0..m}}, d_{x_{1..n}}, d_0, s_{0..255}, x, y) \rightarrow$
$\mathsf{DiscreteLog_G}(d_y, d_{yx}, d_x, d_0, s, x, y)$

For now, $\mathsf{DiscreteLogProofOfKnowledge_G} = \mathsf{DiscreteLog_G}$ (inheriting its preconditions and argument formatting). In the future, another option may be notably more efficient.

$\mathsf{DiscreteLog_G}$ takes the logarithmic derivative of $D(C_0) * D(C_1) * D(C_2) = \prod_{i=0}^{254}(\mu - Z(G_i))$, where D is the divisor, to enable evaluation with a linear combination (instead of hundreds of in-circuit multiplications).

Eagen's work has an x-only divisor equation of $D(C) * D(-C) = \prod_{i=0}^{254}(C.x - G_i.x)$. If we instead take the logarithmic derivative of that, the proof loses the ability to differentiate $G_i.x$ from $(-G_i).x$. We can use this to generate a trinary system $-1, 0, 1$ with 161 powers of G $(G, 3 \cdot G, 9 \cdot G)$ which generates nearly the same 2^{255} group, yet with only 161 trits. Since we cannot distinguish -1 form 1, the prover only has to commit to them as bits, saving 94 in-circuit multiplications. It still satisfies as a proof of knowledge for the discrete log, despite not being binding to the discrete log.

4.5 Circuit

We define two distinct layer gadgets, one for the first layer (which ensures the integrity of the tuple output), and one for all additional layers.

4.5.1 First Layer

- Define $r_o \cdot T, r_c \cdot G, r_j \cdot T$ in-circuit, constrain them on curve, and prove knowledge of the discrete logarithms.
- Define $r_i \cdot U$ in-circuit, constrain it on curve, and prove the bit decomposition of r_i the discrete logarithm of $r_i \cdot U$ over U.
- For the input tuple $\tilde{O}, \tilde{I}, \tilde{C}, R$, prove the x-coordinate of $r_o \cdot T$ distinct from the x-coordinate of $\tilde{O}$, the x-coordinate of $r_i \cdot U$ distinct from the x-coordinate of $\tilde{I}$, the x-coordinate of $r_c \cdot G$ distinct from the x-coordinate of $\tilde{C}$, and the x-coordinate of R distinct from the x-coordinate of $r_j \cdot T$.
- Prove the bit decomposition of r_i (already decomposed and validated as a bitstring) is the discrete logarithm of $R - (r_j \cdot T)$ over V.
- Perform incomplete addition of $\tilde{O}$ with $-(r_o \cdot T)$ to obtain O, $\tilde{I}$ with $-(r_i \cdot U)$ to obtain I, and $\tilde{C}$ with $-(r_c \cdot G)$ to obtain C.
- Prove O, C permissible for hashing.
- Perform tuple set membership with $(O.x, I.x, I.y, C.x)$ within a Pedersen vector commitment. This is the hash for the next layer.

4.5.2 Additional Layer

- Define randomness (r_x, r_y) and constrain it's on curve.
- Prove knowledge of the discrete logarithm of (r_x, r_y).
- Take the blinded Pedersen hash from the prior layer and enforce its x coordinate is inequal to r_x. Then perform incomplete addition with $(r_x, -r_y)$.

  We do not need to check the blinded Pedersen hash is on-curve in-circuit as we can verify that out of circuit.
- With the unblinded Pedersen hash's x, y coordinates, prove it permissible.
- Perform set membership of the unblinded Pedersen's hash x coordinate over the members in a Pedersen Vector Commitment (the new blinded Pedersen hash).

5 Functionality

With the system described, it is important to be clear on the functionality in order to fairly evaluate it.

5.1 Full-Set Privacy

This is the reason for discussing this in the first place.

5.2 Hardware Wallets

Hardware wallets maintain their support due to not needing to perform the membership proof, solely the Generalized Schnorr Protocol after the fact. The Generalized Schnorr Protocol is smaller than the current CLSAG proofs, and is accordingly assumed to be well within the allowed memory footprint. It is only one point larger than the proposed Seraphis ownership proof.

5.3 Multisignature Protocols

Generalized Schnorr Protocols effectively are Schnorr signatures, as the name implies. Accordingly, they benefit from all of the great work performed on Schnorr multisignatures, and can be expected to offer low-complexity, small-constant multisignature protocols.

5.4 Outgoing View Keys

Outgoing view keys, a new private key which enables calculating linking tags for scanned outputs without possession of the private key, are inherently enabled by this proposal. Since x, y are needed to spend, yet the linking tag is solely derived from x, x becomes the shareable outgoing view key. This does require y be unknown to whoever the outgoing view key is shared to, which is not the case for historical outputs (where $y = 0$).

If wallets publish addresses whose public spend key is not $s \cdot G$ yet $o \cdot G + y \cdot T$, where y is the effective private spend key (uniformly sampled), o, the outgoing view key, can be shared. This solely requires wallet software update their key handling internally, and can be done at any moment in time, by any subset of wallets, with no impact to privacy (on- or off-chain) nor network performance.

5.5 Forward Secrecy

We define forward secrecy as an adversary with a discrete log oracle being able to, for any output, find a consistent opening for the given input tuple $(\tilde{O}, \tilde{I}, \tilde{C}, R)$ and its Spend Authorization + Linkability proof.

Per the following Python script,

https://gist.github.com/kayabaNerve/e09ba62d4a6165ced006b037f1068d95

this is true. The script creates an output, decides all of the necessary nonces, and publishes its solutions. The script proceeds to create a random output (specifically, a random linking tag generator for an output), and from there performs extraction of the various witness values/nonces which would have been used. Since it is able to extract a solution for every variable, and rebuild identical commitments, it is able to forge a indistinguishable proof that output was the output spent. Since any output will have an indistinguishable forged proof, one cannot distinguish a legitimate proof.

An adversary with a discrete log oracle also cannot distinguish between an unspent non-forward-secret output and a forward-secret output. Such an adversary can only calculate what the linking tag would be if the output isn't forward secret, and wait to see if that appears (making it a spent non-forward-secret output).

5.6 Transaction Chaining

Since the membership proof does not require knowledge of x, y, the membership proof can be produced after the transaction itself is signed. This would let Alice and Bob form a 2-of-2 multisig, sign a transaction spending an output sent to it (per some mutually known blinding factors), then let Alice or Bob create that output, and once it's past the 10-block lock, publish the transaction spending it without the other party's participation.

We do have to ensure the message the Generalized Schnorr Protocol signs isn't binding to the membership proof however.

6 Integration

With the concepts of the proof established, and the functionality iterated, it's time to discuss how integration will be handled.

6.1 Tree

With the circuit proving an output tuple exists within a Merkle tree, we need to create the tree. The tree will be implemented in C++, with the hash function FFI'd from Rust.

The tree is a l-layer tree where each layer is of alternative widths w_a, w_b. The tree is theoretically considered to be a ∞-layer tree, where all members of branches are zero by default. The tree root is the hash within the lowest branch with only one hash.

For the hash function $\mathcal{H}_{tree}$, parameterized by the layer alternation to $\mathcal{H}_{tree_a}, \mathcal{H}_{tree_b}$, and a list of leaves leaves, the tree is calculated as follows:

1. $c = a$. Hash leaves in w_c chunks with $\mathcal{H}_{tree_c}$ to create $\mathsf{leaves.length}()/w_c$ members of the next layer (denoted $\mathsf{members}$).
2. If $c = a$, $c = b$, else $c = a$. If $\mathsf{members.length}() == 1$, the algorithm terminates. Else, hash $\mathsf{members}$ in w_c chunks with $\mathcal{H}_{tree_c}$ to create $\mathsf{leaves.length}()/w_c$ members of the next layer (which become the new $\mathsf{members}$).
3. Repeat the prior step.

We expose the following external API, minimizing the intricacies of the tree:

- Grow.

  Grow incorporates a list of n leaves (themselves tuples (O, I, C)) into the tree.

- Prune.

  Prune removes a list of n leaves from the tree.

With the tree calculation described, and the API defined, we now detail performant implementations of both functions. We take extensive advantage of how the hashes are additively homomorphic ($\mathcal{H}_{tree_c}([A, 0]) + \mathcal{H}_{tree_c}([0, B]) = \mathcal{H}_{tree_c}([A, B])$).

6.1.1 Grow

For $\mathsf{outputs}$, a set of output tuples (O, I, C), we separate them as

$$\mathsf{outputs_0} = \mathsf{outputs}[..\ \mathsf{w_a} - (\mathsf{tree.length}() \mod \mathsf{w_a})]$$

$$\mathsf{outputs_1} = \mathsf{outputs}[(\mathsf{tree.length}() \mod \mathsf{w_a})\ ..]$$

Add $\mathcal{H}_{tree_c}(([0,0,0,0] * (tree.length() \mod w_a)) + \mathsf{outputs_0}.\mathsf{flatten}())$ to the most recently appended branch hash for the leaves. The flatten operation transforms

$$[(A, B, C), (D, E, F)]$$

to

$$[A.x, B.x, B.y, C.x, D.x, E.x, E.y, F.x]$$

(a series of tuples of points to a series of field elements).

Then, for each w_a chunk of $\mathsf{outputs_1}$, hash the flattened chunk, appending the resulting hash to the next layer.

Please note this technically makes the width of the first layer $4 * w_a$, due to considering four field elements (each one word of the hash function) as individual leaves.

For each branch hash modified from H to H', calculate $H'.x - H.x$, the delta. With the proper alignment and chunking, hash the deltas and add the resulting hash to the proper branch hash on the next layer up. Repeat until the only hash modified is the new tree's root.

6.1.2 Prune

1. For every complete branch of leaves encompassed, remove them.
2. For every branch of leaves partially modified, hash the removed leaves (properly aligned within the hash function) and subtract them from the existing branch hash if the amount of removed leaves is less than the amount of remaining leaves. If the amount of remaining leaves is less, hash those to re-calculate the hash entirely.
3. For every hash whose children were modified, perform the above remove/delta-hash procedure until the new tree's root is updated.

6.2 Permissibility

For an input tuple (O, I, C), O, T must be permissible before they can be hashed into the tree. O has T added until it's permissible, where for $O = x \cdot G + y \cdot T$, y is incremented as many times as T was added. C has G added until it's permissible, where for $C = r \cdot G + a \cdot H$, r (the commitment mask) is incremented as many times as G was added.

The hash functions, outputting points, are programmed to only output permissible points.

6.2.1 Initialization

When an instance of monerod boots with the tree code, it will immediately start indexing every block since genesis. Cryptonote outputs will be accumulated as the key, the linking tag generator, and a Pedersen commitment for the amount (with a randomness of zero). RingCT outputs will be accumulated as the key, the linking tag generator, and their Pedersen commitment.

6.2.2 Normal Operation

When a block achieves a depth of 10 (the constant 10 being from the 10-block lock), all outputs within it are used to grow the tree. On a reorganization exceeding 10 blocks, prune is called for the removed outputs. On a reorganization less than 10 blocks which shortens the length of the chain, prune is called for the outputs within blocks no-longer 10-deep.

6.2.3 Timelocks

If an output has a block-based timelock, it is set to be accumulated with the outputs of that block. If an output has a time-based timelock prior to the activation of the FCMP++ hard fork, it is converted to an estimated block-based timelock and accumulated with the outputs of that block. After the FCMP++ hard fork, time-based timelocks will be rejected entirely.

Preserving time-based timelocks would require defining a linked list where upon block addition, we check if we should add the output at the head of the list (and any further outputs) to the tree. Unfortunately, such an implementation would be quite easy to perform denial-of-service attacks against. We could also define a vector, enabling efficient insertion, yet then we must define and maintain an unbounded global list which at best can be bucketed (though we'd need to prune from the head of the list, shifting the bucket(s)). With block-based timelocks, we still have the unbounded list commentary, yet have efficient topics (the block number) and no partial list additions. Accordingly, one is not worth the headache, and one has a tolerable headache.

6.3 Input Modifications

To minimize modifications to inputs, FCMP++ inputs simply set the amount of key offsets 0.

6.4 Modifications to RingCT Base

We define a new RingCT type, FCMPPlusPlus, and with it a new field, ReferenceBlock. ReferenceBlock is the 32-byte hash of the Monero block which was most recently accumulated into the tree (and therefore must be at least 10 blocks old).

6.5 Modifications to RingCT Prunable

We extend prunable with a byte-buffer which is of fixed-length with regards to the amount of outputs. This byte buffer is entirely left to Rust to interpret.

This byte buffer contains p Generalized Schnorr Protocols (which don't sufficiently benefit from being merged into a single proof).

It also contains n Generalized Bulletproofs, which we can structure in two ways.

- We can minimize bandwidth by sending one Generalized Bulletproof. This would practically mean limiting the amount of transaction inputs to potentially as low as four.

  For context, we limit outputs to sixteen as a transaction with nine outputs has the same amount of computational overhead as a transaction with sixteen outputs. This means for a transaction with nine outputs, we waste seven outputs of computation ($7 * 64$, or 448, multiplications), For inputs with one Generalized Bulletproof, we'd have the exact same effect, except one input of computation is presumably 3072 multiplications. This means a single input of wasted computation is as bad as executing Bulletproof range proofs for sixteen outputs, three times over. for no reason at all.

  With an input limit of four, the most waste which can occur is one input of work by making a three-input transaction.

- We can have no computational overhead by including as many Generalized Bulletproofs as powers of two summed to equal the amount of inputs. For one input, one Generalized Bulletproof. For four inputs, one Generalized Bulletproof. For seven $(4 + 2 + 1)$ inputs, three Generalized Bulletproofs. This doesn't waste any performance regarding verification, and roughly makes the expected bandwidth 2.5 kB per two inputs.

6.6 Modifications to Transaction Verification

ReferenceBlock is confirmed to be on the best chain and 10-blocks deep. The tree root after applying that block is fetched. The Rust code is called with the tree root, the linking tags, the pseudo-outputs, the byte buffer from RingCT prunable, and the hash of the transaction as used for signing.

As of the most recent hard fork (Bulletproofs+), the only fields from prunable which are hashed when signing are the Bulletproofs. That is maintained. RingCT base is modified to exclude ReferenceBlock when being serialized for the hash for signing. While this would imply ReferenceBlock should be placed within prunable, this field cannot be pruned (as it's necessary to evaluate if the transaction should be dropped upon deep reorganization). The lack of hashing it, and malleability regarding the tree used, is explicitly allowed in order to support transaction chaining.

6.7 Modifications to Addresses

Addresses do not need modifications. All existing addresses will continue to work as-is. Wallets will work without modifying their keys at all.

6.7.1 Outgoing View Keys

In order to take advantage of the outgoing view key functionality, changes to the wallet's internals must be made. This is completely optional to the wallet and can be done at any time without impacting privacy. If it required generating a new address type, that would segment users (impacting privacy), hence the explicit statement there is no impact to privacy.

Wallets would not only generate the new private key, y, they would also design and develop a format for sharing x, the outgoing view key, and update software to calculate linking tags when scanning outputs if they had the incoming view key and x. They would also specify their address with the public spend key $x \cdot G + y \cdot T$, instead of $x \cdot G$. This change is indistinguishable to a recipient of an address and indistinguishable to an adversary with a discrete log oracle.

6.7.2 Forward Secrecy

Forward secrecy, if enabled on the protocol level, requires the modifications for outgoing view keys. No additional modifications are required.

6.8 Modifications to the RPC

The RPC is extended with a route to return the path for an output (by index) within a specified tree. The existing decoy routes are maintained. A new distribution is added comprehensive to both the Cryptonote outputs and the RingCT outputs.

6.9 Modifications to Wallets

Wallets fetch the unified distribution, yet do not fetch the decoy information. Instead, they fetch the path of each selected decoy (and the actual output).

Alternatively, wallets may locally build the tree while scanning the blockchain. This removes the need to request paths for any outputs, as wallets may now locally request the path for their output.

Instead of calling CLSAG's prove with the decoy information, wallets would call the FCMP++ prove with the path of the output being spent. No other changes to wallets are required for full-chain membership proofs (solely for additional, wallet-optional functionality, for which the changes can be done at any time).

6.10 Multisignature Wallets

Existing Monero multisignature wallets are still usable. We maintain the DKG process, which effectively creates a n-of-n multisig. We also do not touch wallet scanning/state management, nor utilities (linking proofs).

Multi-party transaction construction isn't modified. The actual signing code is modified. The existing CLSAG code is removed, replaced with either a C++ implementation of multisig OR calls to Rust multisignature code. The latter would involve passing the multisignature wallet key data from C++, reformatting into a structure the Rust code can work with, and obtaining the preprocesses/signature shares from Rust. The communication and incorporation of those into the signed transaction would occur as prior handled.

7 Future

Monero can deploy Seraphis, the codebase, in the future. The improvements to transaction construction, and sender-receiver communication (regarding one-time keys and shared secrets), are well-reasoned for Monero and desirable.

Monero can also still deploy Seraphis, the linking tag format, requiring a migration and new addresses. That may offer more efficient membership proofs/a better choice of elliptic curves (better choice throughout the project, not just regarding membership). Generalized Bulletproofs, the gadgets described, and the additional layer circuit section would all be used for a deployment of FCMPs with Seraphis. We'd solely have to simplify the first layer circuit section. The usage of Generalized Schnorr Protocols has also been proposed with Seraphis, which would make an implementation of them from this work potentially reusable.

JAMTIS can also still be deployed (over FCMP++s or over Seraphis).