Implementing Seraphis (WIP)*

Draft v0.0.2†

koe `ukoe@protonmail.com`

April 29, 2023

Abstract

We discuss how Seraphis, a transaction protocol abstraction for p2p electronic cash systems, may be implemented. Specifically, we specify a set of elliptic curve generators, describe a concrete balance proof, explain how coinbase enotes and transaction fees may be integrated into a Seraphis transaction protocol, show how the addressing scheme Jamtis satisfies requirements set by the Seraphis abstraction, provide recommendations for optimizing a transaction protocol design, and explore nuances related to the process of constructing transactions. We also introduce a comprehensive set of information proofs for Jamtis-based transactions.

1 Introduction

Seraphis [9] is a transaction protocol abstraction, and hence does not specify any concrete implementation details. To that end, we discuss how to implement various components of a Seraphis-compatible protocol. The highlights of our design recommendations are as follows.

1. Jamtis [18] is a Seraphis-compatible addressing scheme that enables full balance recovery without the master spend key, supports a third-party balance recovery process with substantially better privacy than third-party scanning with CryptoNote view keys, eliminates the 'duplicate onetime address' and 'subaddress look-ahead' issues that plague the CryptoNote addressing scheme, and provides conditional forward-secrecy against a DLP-solver (e.g. quantum adversary).
2. Membership proofs may be constructed after a transaction has been signed. This makes multisignatures simpler and more robust, and enables 'transaction chaining' where it is possible to construct a partial transaction B that spends an enote from transaction A before A has been added to the ledger (Section 10).

Appendix A presents a comprehensive set of information proofs for Jamtis-based Seraphis transactions (e.g. address ownership proofs, reserve proofs, etc.). Appendix B presents a flexible Schnorr-like proof for key matrices constructed from sets of private keys and base keys, which is used in the information proofs.

*

†This is just a draft, so it may not always be available wherever it is currently hosted.

1.1 Acknowledgements

We would like to thank tevador for his work as the lead author of Jamtis [18], Justin Berman and Luke Parker for their feedback and contributions to Jamtis, and dangerousfreedom for his contributions to the Jamtis information proofs.

2 Preliminaries

2.1 Public parameters

[1]Let $\mathbb{G}$ be a cyclic group of prime order $l > 3$ in which the discrete logarithm problem is hard and the decisional and inverse decisional Diffie-Hellman assumptions hold, and let $\mathbb{Z}_l$ be its scalar field. Let $\mathcal{H} : [0,1]^*$ and $\mathcal{H}^n[0,1]^* \to \mathbb{Z}_l$ be cryptographic hash functions. We add a subscript to hash functions, such as $\mathcal{H}_{src}$, in lieu of domain-separating explicitly; any domain-separation method may be used in practice (e.g. an ASCII string corresponding to a domain-separated use case, such as $\mathcal{H}(\text{“}\textit{sender\_receiver\_secret}\text{”} \;||\; [\text{hash input}])$).

We define four distinct generators $G, H, X, U \in \mathbb{G}$ that map to generators from the Seraphis paper. The generator G should be the ‘main’ generator of $\mathbb{G}$ according to the relevant convention, and the other generators should be produced using public randomness.

$$\begin{aligned} G &= G_0 = H_0 \\ H &= H_1 \\ X &= G_1 \\ U &= G_2 = J \end{aligned}$$

We also define an isometric cipher and decipher functions `Ciph` and `Deciph` [terminology? notation?].

All public parameters are assumed to comprise a global reference string known to all players. For readability, we generally exclude explicit reference to public parameters in algorithm definitions and Fiat-Shamir transcript hashes.

2.2 Notation

- We use additive notation for group operations on $\mathbb{G}$. This means, for example, that the binary group operation between G and H is denoted $G + H$.
- This paper contains no exponentiation unless explicitly stated. Superscripts such as the o in k^o are in most cases merely for descriptive purposes and have no mathematical significance.

[1] Parts of this section were copied mostly verbatim from the Triptych preprint [15].

- For group element P and scalar $x \in \mathbb{Z}_l$, xP and $x * P$ both indicate scalar multiplication. The use of asterisks ($*$) in some places but not others is meant to aid visual clarity where appropriate (usually when multiplying by a parenthesized scalar or by a scalar that has a superscript).
- Modular multiplicative inverse group operations use the notation $(1/x) * P$.
- Tuples are indicated with brackets, e.g. $[A, B, C]$. To avoid confusion, we always explicitly refer to tuples as tuples wherever they appear (e.g. 'the tuple $[A, B, C]$').

2.3 Squashed enote model

We use {$\underline{\text{this syntax}}$} to highlight text that is specific to the Seraphis squashed enote model.

3 Applying the generators

Using the generators defined in Section 2.2, an enote will look like this:

- **Amount commitment**: $C = xG + aH$
- **Address**: $K^o = k_0 * G + k_1 * X + k_2 * U$
- **Memo**: An arbitrary memo field.

We use Jamtis to specify how an enote should be constructed in Section 8.3.

An enote-image will look like this:

- **Masked commitment**: $C' = (t_c + x) * G + aH$
- **Masked address**: $K' = (t_k + k_0) * G + k_1 * X + k_2 * U$
- **Key image**[2]: $\tilde{K} = (k_2/k_1) * U$

4 Balance proofs

For amount balance proofs, we use a non-zero 'blinding factor remainder' p_r, and publish it inside transactions.[3] Using $p_r > 0$ avoids requiring interdependency between enote images. Any range proof construction may be used that satisfies [todo: xyz requirements] (e.g. Bulletproofs+ [2]).

[2] A key image is a style of 'linking tag' characterized by being derived from an enote's address key.

[3] An alternative to adding p_r to transactions explicitly would be creating a signature on the public key p_rG (as done in Lelantus-Spark [7]). This can prevent problem 2 from Section 4.3. However, if proof structures resilient to problem 2 are used, we do not think signing p_rG offers any advantages (problem 1 is unrelated to p_r).

4.1 Constructing a balance proof

Suppose a transaction spends $j \in 1, ..., m$ old enotes and creates $t \in 1, ..., p$ new ones. A balance proof involves the following steps.

1. Let the masked commitments in enote-images be denoted $C'_j = v_{c,j}G + a_j H$. Let the commitments in new enotes be denoted $C_t = y_t G + b_t H$.
2. For $j \in 1, ..., m$, randomly select $v_{c,j} \in_R \mathbb{Z}_l$. For $t \in 1, ..., p$, randomly select $y_t \in_R \mathbb{Z}_l$.
3. Compute $p_r = [\sum_{j=1}^{m} v_{c,j}] - [\sum_{t=1}^{p} y_t]$.
4. Produce a range proof for each new amount commitment C_t $\{\underline{\text{and each masked amount commitment } C'_j}\}$ for the allowed amount range $[0, 2^z)$.[4]

Record all $\{C'_j, C_t, p_r, \text{range proofs}\}$ in a transaction.

Note: As required by Seraphis, the values $t_{c,j} = v_{c,j} - x_j$ will be uniformly distributed because $v_{c,j}$ are generated randomly and x_j are former y_t values that were also randomly generated.

4.2 Verifying a balance proof

To verify a balance proof, perform the following steps.

1. Check that $\sum C'_j == \sum C_t + p_r$ holds.
2. Check that each C_t $\{\underline{\text{and each } C'_j}\}$ has a valid range proof.

If the above checks hold for the transaction, then, within a security factor, there must be a balance on generators G and H in the two commitment sets (input enote-image commitments and output enote commitments).

In conclusion, the amounts must balance between input and output enotes.

4.3 Sender-receiver anonymity

If a transaction only has one input ($m = 1$) and all its outputs' y_t are known by an observer (e.g. because they received all enotes produced by the transaction), then the observer will know the value $v_{c,1} = \sum_{t=1}^{p} y_t + p_r$.

However, even if the observer is the original sender of the enote that the transaction author is spending, they won't necessarily know any more information about the transaction's input than if they weren't the original sender.

[4] The value z must be lower than the order l of $\mathbb{G}$ minus a security factor $k < \log_2(l)$: $0 \leq z < \log_2(l) - k$.

First note that the observer, by knowing all y_t, will presumably also know the total amount output by the transaction (assuming they know the transaction fee, if relevant), and hence will know the input amount a_1.

Second, even if they were the original sender, the input could have been sent to the transaction author by someone else. Despite knowing both x_1 and $v_{c,1}$, the observer has no way to know if the real input actually had a different blinding factor x'_1, since t_c is uniformly distributed at random.

There are two problems to consider.

1. If the amount a_1 is 'unusual' (i.e. unlikely to have been created by someone else), then the observer can guess with high probability of success that they created the enote being spent, assuming that enote was referenced by the input's membership proof. This problem may extend to multi-input transactions if the 'low bits' of the total amount are unusual (e.g. because one input has a fingerprint recorded in low bits of its amount value, and other inputs' amounts have low bits set to zero).

   Even if the amount isn't unusual, if the anonymity set size of membership proofs is relatively small, then there is a very low probability that the observer's enote was randomly selected as a decoy and just happened to have the same amount as the real enote being spent.

2. If $t_{c,1}$ is used as a secret input to a proof (e.g. a discrete log proof of the commitment to zero $C' - C$ with respect to G), then the observer may be able to guess and check the proof structure with known values of x_1 to see if $t_{c,1} = \sum_{t=1}^{p} y_t + p_r - x_1$ is in fact that secret input (depending on the proof structure used).[5]

Both problems are mitigated or solved by including a 'change enote' in each transaction, even if its amount must be zero.[6] A change enote is an enote the transaction author sends to himself if the total output amount of his transaction exceeds the amount he intends to send to other people (unavoidable if no combination of owned enotes' amounts equals the intended total output amount of his transaction).

5 Input proofs

Membership proofs may be implemented with any cryptographic proof that satisfies the Seraphis protocol's requirements. For example, a Grootle proof [9] in the squashed enote model, or a Lelantus-Spark one-of-many proof [7] in the base enote model.

Ownership/unspentness proofs may likewise be implemented with any proof satisfying the Seraphis protocol's requirements. For example, a Seraphis composition proof [9].

[5] For an example of where this can be a problem, CLSAG [6] and Triptych [15] both require 'extra' key images computed like $t_c P$ (where P is public information). This means if the observer knows the input commitment blinding factor (and all output commitment blinding factors), then they can identify the true spend of a 1-input transaction via guess-and-check.

[6] There are niche cases where the first problem is unsolvable. For example, the sender could allow a 'low bit' fingerprint to propagate from an input to an output. The observer may also be able to infer, by the mere fact an enote he created was referenced by a membership proof, that his enote is being spent.

6 Coinbase enotes

For a cryptocurrency to be widely adopted, observers should be able to verify that the total supply of money matches their expectations based on coinbase enotes and transactions recorded in the ledger.[7]

Let coinbase enotes have a special format. Instead of recording amount commitments, they should record the amounts in cleartext. For a coinbase enote to be referenced in a membership proof's input set $\mathbb{S}$, it must be 'converted' into a normal enote first.

Converting a coinbase enote to a normal enote is very simple.

- Set the enote's address equal to the coinbase enote's address: $K^o_{enote} = K^o_{coinbase}$.
- Set the enote's commitment equal to an unmasked commitment to the coinbase enote's amount a: $C_{enote} = aH$.

When a transaction's membership proof references enotes in the ledger, it is common to reference them by index. Verifiers look up those indices, then $\{\underline{\text{squash and}}\}$ copy the enotes they find into $\mathbb{S}$. If a verifier finds a coinbase enote at a lookup index, they should convert it into a normal enote before $\{\underline{\text{squashing it and}}\}$ copying it into $\mathbb{S}$.[8]

If a transaction spends a coinbase enote, then its enote-image's masked amount commitment will hide the amount involved even though the original amount had no blinding factor.

7 Transaction fees

Most cryptocurrencies have a 'transaction fee'. Each transaction must send a small fee to a third-party. Fees disincentivize creating large numbers of transactions that excessively burden the network and node operators. They also allow transaction authors to prioritize their transactions. Transactions with high fees will typically be added to the ledger faster than those with low fees if the p2p network is congested.

To ensure fees are publicly verifiable, they are usually recorded in cleartext in transactions. Fees are then converted into enotes and added to the ledger at a later date. The rules around this conversion process are minutiae defined by each cryptocurrency.[9]

Fees must be incorporated into amount balances. Transaction verifiers can use the following simple procedure.

[7] Observers should also expect that coinbase enotes only appear in the ledger when well-defined rules have been satisfied (e.g. they were created in the genesis block, or via PoW/PoS mining).

[8] In practice, transaction verifiers can store converted coinbase enotes directly in/alongside a local copy of the ledger, so they don't have to be converted each time they are referenced by a transaction.

[9] In PoW cryptocurrencies, a block's miner typically adds the fee amounts from their block's transactions into their coinbase enotes (the outputs of their 'miner transaction') as part of their block reward (which usually includes newly minted money).

1. Convert a transaction's fee amount f into an unmasked commitment: fH. Require that $0 \leq f < 2^z$.

2. Test that the transaction's amounts balance:

$$\sum_j C' \stackrel{?}{=} \sum_t C + p_r + fH$$

8 Jamtis

In cryptocurrencies, money may be transferred to another person by submitting a transaction to the p2p network. The transaction should spend money owned by the transaction author and contain new enotes owned by the recipients of that money. Ownership is implemented by constructing enotes using the 'addresses' of intended owners. An enote can only be spent when the owning address's private keys are known. Jamtis [18] is a Seraphis-compatible address scheme with various useful properties that will be explored throughout this section.

8.1 Jamtis private keys and public addresses

Jamtis addresses are derived from a hierarchy of private keys. The utility of this hierarchy will be discussed in Section 8.6.

8.1.1 Jamtis core

Jamtis begins with two root private keys.[10]

- **Master key**: $k_m \in_R \mathbb{Z}_l$
- **View-balance key**: $k_{vb} \in_R \mathbb{Z}_l$

From the view-balance key we derive the following:[11]

- **Find-received key**: $k_{fr} = \mathcal{H}^n_{fr}(k_{vb})$

[10] The view-balance key is not required to be derived from the master key for two primary reasons. Firstly, users may have pre-existing key pairs that they want to re-use in Seraphis. Secondly, in multisignature schemes it is not feasible to derive anything from the master key since no single party knows its full value.

[11] The Jamtis keys $k_{fr}, k_{ua}, K_2^j, K_3^j$ may be defined on a separate elliptic curve from $\mathbb{G}$, for example with X25519 for maximized ECDH performance. For simplicity in this document we assume the primary curve $\mathbb{G}$ is used. If those keys are defined on a group with cofactor $h > 1$, then the cofactor should be included as a scalar multiplier in sender receiver secrets, for example $K_d = h * k_{fr} * K_e$. If this is not done, then a malicious transaction author can determine if the recipient's relevant ECDH secret key is a multiple of h (or any divisor of h) or not. They only need to set, for example, $K_e = K_e + K^{hf}$, where K^{hf} is a point in the subgroup of order h (or a factor of h). If the recipient successfully performs balance recovery on the enote and notifies the sender, then the sender will know that k_{fr} is a multiple of the subgroup order of K^{hf}. This is because $k_{fr} * K^{hf} == I$ when k_{fr} is a multiple of K^{hf}'s subgroup's order, allowing the recipient to successfully reproduce K_d only in that case. Including h in sender-receiver secrets means cofactor-subgroup points will always be 'canceled out' if they are present in ECDH base keys.

- **Unlock-amounts key**: $k_{ua} = \mathcal{H}^n_{ua}(k_{vb})$
- **Generate-address secret**: $s_{ga} = \mathcal{H}_{ga}(k_{vb})$
- **Cipher-tag secret**: $s_{ct} = \mathcal{H}_{ct}(s_{ga})$

We now compute the base public keys from which Jamtis addresses will be built:

- **Base spend pubkey**: $K_s = k_{vb}X + k_m U$
- **Unlock-amounts pubkey**: $K_{ua} = k_{ua}G$
- **Find-received pubkey**: $K_{fr} = k_{fr}K_{ua}$

8.1.2 Jamtis public addresses

Jamtis public addresses are derived from the Jamtis private keys and base public keys, and a user-defined ‘address index’ $j \in \{0,1\}*$. The address index is ciphered and included alongside the public address keys.

At the heart of a Jamtis public address is an ‘address-index generator’ secret:

$$s^j_{gen} = \mathcal{H}_{gen}(s_{ga}||j)$$

From the generator secret we derive three spendkey extensions

$$k^j_g = \mathcal{H}^n_{seg}(K_s||j||s^j_{gen})$$
$$k^j_x = \mathcal{H}^n_{sex}(K_s||j||s^j_{gen})$$
$$k^j_u = \mathcal{H}^n_{seu}(K_s||j||s^j_{gen})$$

and one address private key:

$$k^j_a = \mathcal{H}^n_{ap}(K_s||j||s^j_{gen})$$

The rationale for binding to j twice (directly and via s^j_{gen}) and to K_s is discussed in Appendix A.2.

We define the Jamtis public address keys as follows:

$$K^j_1 = k^j_g G + k^j_x X + k^j_u U + K_s$$
$$K^j_2 = k^j_a * K_{fr}$$
$$K^j_3 = k^j_a * K_{ua}$$

The index is ciphered as $c^j = \texttt{Ciph}[s_{ct}](j)$ and prepended to a ‘decipher hint’ to give the address tag:[12]

$$\texttt{addr\_tag}^j = c^j \ || \ \mathcal{H}_{dch}(s_{ct}||c^j)$$

A full Jamtis public address is the tuple $[K^j_1, K^j_2, K^j_3, \texttt{addr\_tag}^j]$.

[12] The Jamtis address tag decipher hint only needs to be 1-2 bytes in size to be effective.

8.2 Jamtis sender-receiver secret derivation

Enote owners would like to discover the enotes they own in the ledger, read the amounts in those enotes, reconstruct commitments in order to perform balance proofs in new transactions, and learn enough secret key material in enote addresses so they can construct key images.

The answer first pioneered by CryptoNote [20] for privacy-focused transaction protocols depends on a Diffie-Hellman shared secret between the sender and receiver of an enote. Jamtis evolves that approach by using one Diffie-Hellman shared secret K_d and two high-level sender-receiver shared secrets s_1^{sr}, s_2^{sr}. The two high-level secrets are derived together using two separate derivation paths depending on the enote type.

8.2.1 Diffie-Hellman derived key K_d

At the core of Jamtis enote construction is an enote ephemeral private key $r \in_R \mathbb{Z}_l$ and associated enote ephemeral public key K_e. The value K_e is recorded alongside a Jamtis enote for use by the enote's owner.

As we will see, an enote owner can always use their find-received key k_{fr} with K_e to recover the Diffie-Hellman derived key K_d:

$$K_d = k_{fr} K_e$$

There are two possible ways to compute K_e and K_d during transaction construction.

Standard

For most enotes, K_e and K_d will be computed directly from the recipient's address keys.

$$K_e = r K_3^j$$
$$K_d = r K_2^j$$

In this case, $K_d = rK_2^j = rk_{fr}K_3^j = k_{fr}K_e$.

Shared-K_e optimization

If a transaction contains one recipient and one additional enote sending funds to the transaction author (e.g. a leftover/change amount from the difference between inputs, the one recipient, and the transaction fee), then that additional 'selfsend' enote can share the other enote's K_e.

In this case, the transaction author knows his own k_{fr} key, so he can define the following:

$$K_e^{\text{self}} = K_e^{\text{other}}$$
$$K_d = k_{fr} * K_e^{\text{self}}$$

which ensures he can properly recompute K_d during balance recovery.

A transaction that uses the shared-K_e optimization only needs to store one copy of K_e.[13]

[13] A major advantage of sharing K_e is you only need to derive K_d once instead of twice. If most transactions in the ledger have two outputs, and all two-output transactions use the shared-K_e optimization, then there will be a substantial reduction in K_d computations relative to a scenario with no K_e sharing.

8.2.2 Secret uniqueness: `input_context`

To ensure uniqueness of enote components between transactions, we will bind the secret s_1^{sr} to the 'input context' of the transaction it will be used in. Transaction inputs are never repeated in the ledger, which guarantees uniqueness of s_1^{sr} even if r is repeated.

- **Coinbase txs**: $\texttt{input\_context} = \mathcal{H}_{icc}(\texttt{block\_height})$
- **Normal txs**: $\texttt{input\_context} = \mathcal{H}_{icn}(\tilde{K}_1 || ... || \tilde{K}_m)$

To further ensure uniqueness *within* a transaction, transaction verifiers must mandate that all values K_e in a transaction are unique (with a further caveat for selfsend enotes mentioned in Section 8.2.4).[14]

8.2.3 Secrets s_1^{sr}, s_2^{sr} (derivation path 1): normal enotes

For normal enotes that transfer funds to another person, the two sender-receiver secrets s_1^{sr}, s_2^{sr} are derived from K_d and r.

Sender

$$\begin{aligned} s_1^{sr} &= \mathcal{H}_{sr1n}(K_d || K_e || \texttt{input\_context}) \\ s_2^{sr} &= \mathcal{H}_{sr2n}(r * G) \end{aligned}$$

Recipient

$$\begin{aligned} s_1^{sr} &= \mathcal{H}_{sr1n}(K_d || K_e || \texttt{input\_context}) \\ s_2^{sr} &= \mathcal{H}_{sr2n}((1/(k_a^j * k_{ua})) * K_e) \end{aligned}$$

The second secret s_2^{sr} is computed from an 'inverse' Diffie-Hellman exchange involving the recipient's address key $K_3^j = (k_a^j * k_{ua}) * G$. The usefulness of s_2^{sr} will be explored in Sections 8.4.3 and 8.6.

8.2.4 Secrets s_1^{sr}, s_2^{sr} (derivation path 2): selfsend enotes

When sending funds to himself, a transaction author derives the two sender-receiver secrets s_1^{sr}, s_2^{sr} from K_e and his Jamtis key k_{vb}. The secrets can be recomputed in the same way during balance recovery.

As we will see in Section 8.6.1, providing a separate derivation path for selfsend enotes improves the privacy attributes of balance recovery when k_{fr} is given to a semi-trusted third party.

It is possible to define multiple different selfsend enote types (e.g. change enotes, self-spend enotes, dummy selfsend enotes). For convenience, we domain-separate s_1^{sr} based on the different selfsend

[14] It is safe for multiple K_e in a transaction to be cofactor-variants of the same point, since we bind s_1^{sr} to the byte-wise representation of K_e.

enote types τ.

$$s_1^{sr} = \mathcal{H}_{sr1s[\tau]}(k_{vb}||K_e||\texttt{input\_context})$$
$$s_2^{sr} = \mathcal{H}_{sr2s}(k_{vb}||s_1^{sr})$$

In this case s_2^{sr} does not rely on the inverse Diffie-Hellman secret rG, which simplifies the shared-K_e optimization because there is no need for the value $(1/(k_a^j * k_{ua})) * K_e^{\text{other}}$ when building the shared-K_e selfsend enote.

Uniqueness: To ensure uniqueness of s_1^{sr} and s_2^{sr} when K_e is shared between two enotes, transaction authors must avoid sharing K_e between two selfsend enotes of the same type. The secrets will be unique in all other cases since the shared-K_e enote is a selfsend, and each selfsend enote type's secrets are domain-separated from secrets of all other enotes types.

Optimized design: Selfsend enote secrets can be computed with three hash operations (including the input context), whereas normal enote secrets require three hash operations, one scalar inversion, and two ECDH exchanges.

8.3 Jamtis enote construction

Jamtis enotes are constructed in a linear fashion from K_d, s_1^{sr}, and s_2^{sr}.

8.3.1 Amount commitment and encoded amount

In order for an enote's owner to recover the enote amount a and recompute the amount blinding factor x, we encode the amount and store it inside the enote along with the amount commitment C, and compute x deterministically.

$$C = \mathcal{H}_{bf}^n(s_1^{sr}||s_2^{sr}) * G + a * H$$
$$a_{enc} = a \texttt{ XOR } \mathcal{H}_{ea}(s_1^{sr}||s_2^{sr})$$

where $x = \mathcal{H}_{bf}^n(s_1^{sr}||s_2^{sr})$ can be recomputed by the enote's owner.

8.3.2 Onetime address

Enote addresses are called 'onetime addresses' because they can only be used to produce one key image, and hence should only appear once in the ledger. An enote's onetime address is:

$$\begin{aligned} K^o = \mathcal{H}_{kog}^n(K_1^j||s_1^{sr}||C) * G + \\ \mathcal{H}_{kox}^n(K_1^j||s_1^{sr}||C) * X + \\ \mathcal{H}_{kou}^n(K_1^j||s_1^{sr}||C) * U + K_1^j \end{aligned}$$

We will justify the use of three extensions in Section 8.7. We explain why binding the extensions to K_1^j is important in Appendix A.3. Onetime addresses do *not* bind to s_2^{sr} so they may be efficiently recovered with only one ECDH exchange using k_{fr} (see Section 8.5).

Binding to C for uniqueness

Onetime addresses are bound to an `input_context` via s^{sr}_1 so that they will be unique in the ledger, however acquiring an input context requires *access* to the ledger. If access to the ledger is intermediated by a semi-trusted third party, it is possible for that party to send false input contexts to a person engaging in balance recovery. If that third party collaborates with a transaction author, they could cause the person doing balance recovery to identify owned enotes with duplicate onetime addresses. To mitigate that risk, onetime addresses are bound to the amount commitment C, ensuring that even if a person is using falsified input contexts during balance recovery, they will only identify duplicate-K^o enotes that have the exact same amounts. This makes it safe to completely ignore all but the oldest copy of the enote.[15],[16]

8.3.3 Encrypted address tag

To facilitate balance recovery, an encrypted version of the Jamtis address tag is recorded in each enote.

$$\texttt{addr\_tag\_enc} = \texttt{addr\_tag XOR } \mathcal{H}_{ate}(s^{sr}_1 || K^o)$$

We bind to K^o for robustness.[17]

8.3.4 View tag

As a balance recovery optimization, we include a MAC-like hash of K_d called a 'view tag' in enotes [19]. A person doing balance recovery can recompute the view tag for a given enote and check it against the stored view tag. If the view tags don't match then all other balance recovery steps can be skipped (the other steps are quite expensive in total, compared to the ECDH exchange for K_d).[18]

$$\texttt{view\_tag} = \mathcal{H}_{vt}(K_d || K^o)$$

We bind to K^o in order to transitively bind to all other components of an enote, ensuring the view tag derivation will never be repeated between two enote construction events.

8.3.5 Enote summary

A Jamtis enote sending funds to an address $[K^j_1, K^j_2, K^j_3, \texttt{addr\_tag}^j]$ is therefore the tuple[19]

[15] If enotes are associated with memos containing critical information, then ignoring newer duplicate-K^o enotes may not be safe.

[16] We keep track of the oldest copy instead of the newest copy in case of ledger reorganizations that remove newer copies but leave older ones untouched.

[17] It is feasible, although unlikely, that s^{sr}_1 could be re-used between separate transaction attempts using different recipients. For example, a shared-K_e selfsend could use two different selfsend addresses but the same original recipient and same enote ephemeral private key r. In that case, binding to K^o ensures different address tags won't be encrypted using the same secret.

[18] A view tag only needs to be 1 or 2 bytes to effectively amortize post-view-tag computations.

[19] A Jamtis coinbase enote will record a explicitly instead of C and a_{enc}.

$$[K^o, C, a_{enc}, \texttt{addr\_tag\_enc}, \texttt{view\_tag}]$$

The enote is associated with an ephemeral public key K_e and an `input_context`.

8.4 CryptoNote address scheme flaws

The Jamtis design we just presented fixes four important flaws in the design of the CryptoNote address scheme.

8.4.1 Onetime address duplication

CryptoNote enotes in the ledger may contain duplicate onetime addresses even though only one of them may be spent. There are two problems with that situation [8].

1. User wallets must track all duplicates and only consider the enotes with the highest amounts to be spendable. Otherwise users could be tricked into 'burning' their own funds.
2. Protocols implemented on top of a cryptocurrency with CryptoNote addresses, such as multisignature schemes or atomic swaps, must take extra precautions to ensure money transfers always guarantee spendability of sent funds.

Jamtis solves the duplication issue by baking an `input_context` into enotes, as discussed in Section 8.2.2.[20]

8.4.2 Subaddress lookahead

A CryptoNote user may generate many 'subaddresses' [14] from their CryptoNote private keys, in addition to their main CryptoNote public address.[21] Much like Jamtis addresses, each CryptoNote subaddress is derived from a user-defined index.

In CryptoNote balance recovery [20, 10], users detect an owned enote by 'unwrapping' the onetime address to examine the underlying spend key, and then matching that unwrapped key against a table of precomputed subaddresses (plus the main address). If there is a match, then the user owns the enote. However, if an enote is owned by a subaddress not in the precomputed table, then the enote will not be identified as owned. As a consequence, CryptoNote balance recovery is error-prone and it is not feasible to randomly generate subaddresses.

Jamtis encrypted address tags cleanly solve this issue by enabling users to directly recover the address indices of owned enotes during balance recovery. If the index j is sufficiently large then addresses can be randomly generated with only negligible risk of collisions.[22]

[20] A naive solution to prevent duplicate onetime addresses would be to simply ban duplicates from appearing in the ledger. Doing that, however, would enable malicious actors in the network to 'kill' in-flight transactions by creating malicious transactions containing the honest transactions' enotes' onetime addresses. If the malicious transactions are confirmed into the ledger then the honest transactions would become invalid.

[21] Subaddresses are an extension on top of CryptoNote, but we consider them part of CryptoNote for simplicity.

[22] An index space of 128 bits is considered sufficiently large for most use-cases [12].

8.4.3 Janus attack

CryptoNote balance recovery involves a two-part process. First is an ECDH exchange using the CryptoNote view key to get a shared secret (much like K_d). Then the shared secret is used to unwrap the spend key of the address that owns the enote (similar to K_1^j).

Since that two-part process is separated, it is possible for each part to be constructed from a different subaddress. A transaction author can optimistically make an enote from two subaddresses they suspect are owned by the same person. Their theory will be confirmed if that person notifies the author they received the enote. This is called a 'Janus attack' [4].

Jamtis solves this by binding enote contents to the index j of the recipient address (see Section 8.5 for the entire balance recovery process).

1. The enote ephemeral key K_e depends on k_a^j: $K_e = rK_3^j = r * (k_a^j * k_{ua}) * G$
2. The recipient recovers index j by decrypting `addr_tag_enc` with $K_d = k_{fr}K_e$ (and then deciphering the decrypted address tag).
3. The recipient recomputes onetime address K^o using K_s, j, and K_d.
4. The recipient computes the second secret s_2^{sr} from the value $(1/(k_a^j * k_{ua})) * K_e$.

The key here is in the last point. Specifically, s_2^{sr} depends on both K_e and the recovered j. If the recovered j does not match the j' of the key $K_3^{j'}$ used to construct $K_e = rK_3^{j'}$, then the user will compute a different s_2^{sr} from the transaction author.

In detail, the user will compute s_2^{sr} with the ECDH key $(1/(k_a^j * k_{ua})) * K_e = r * (k_a^{j'}/k_a^j) * G$, which the transaction author cannot know since k_a^j and $k_a^{j'}$ are secrets. Therefore if a user recovers s_2^{sr} successfully (which can be verified by reproducing C correctly), and assuming K^o was recomputed properly, then the transaction author must have used address components derived from the same index j.

The Janus attack is not an issue for Jamtis selfsend enotes since they are constructed with the view-balance key. A person with the view-balance key already has the power to generate all Jamtis addresses and decipher the address tags of existing addresses.

8.4.4 Incomplete view key

Due to how CryptoNote key images are defined [20], they can only be computed using both CryptoNote private keys (the view and spend keys). This means a user cannot use their CryptoNote view key in isolation to identify when their enotes are spent (by computing key images and checking if they are present in the ledger). As such, the view key only has partial view access to a user's balance.

In contrast, Seraphis key images can be computed using the Jamtis view-balance key k_{vb} in combination with the public key $k_m U$.[23]

8.5 Jamtis balance recovery

Jamtis balance recovery can be divided into two steps: view tag checks and enote recovery. Enote recovery can occur via the 'normal' or 'selfsend' paths, which are very similar.

8.5.1 View tag check and K_d

Given an enote with associated enote ephemeral key K_e and `input_context`, a user who wants to see if they own the enote begins by checking its view tag. This requires the user's k_{fr}.

1. Compute the nominal key derivation: $K'_d = k_{fr} * K_e$
2. Compute the nominal view tag: $\texttt{view\_tag}' = \mathcal{H}_{vt}(K'_d || K^o)$
3. If $\texttt{view\_tag}' \neq \texttt{view\_tag}$ then $\texttt{ABORT}$.

8.5.2 Full enote recovery

Once an enote has been flagged as 'possibly owned', full enote recovery can proceed. Normal scanning will use the key derivation K'_d, while selfsend scanning will test a trial selfsend type τ.

In selfsend scanning, if one τ fails then other values of τ can be tried. Selfsend τ checks are cheap if the address tag hint is sufficiently large because only two hashes are required to test it.

The user first computes s^{sr}_1 and recovers the address index j.

- **Normal scan**: This requires K'_d and the user's s_{ct}.
- **Selfsend scan**: This requires the user's k_{vb} and s_{ct}.

1. Compute the first enote secret
    - **Normal scan**: $s'^{sr}_1 = \mathcal{H}_{sr1n}(K'_d || K_e || \texttt{input\_context})$
    - **Selfsend scan**: $s'^{sr}_1 = \mathcal{H}_{sr1s[\tau]}(k_{vb} || K_e || \texttt{input\_context})$

[23] In an early version of the Seraphis paper, linking tags had the form $\tilde{K} = (1/(\mathcal{H}(s^{sr}_1) + k_{vb})) * G$. This meant a view-balance wallet could demonstrate the discrete log of $\tilde{K}$ with respect to G, allowing them to look at an enote received to the underlying master wallet, extract the linking tag, send a new enote to themselves with the same linking tag, then spend that enote so the linking tag is added to the ledger. The original enote seen by the view-balance wallet would be unspendable (i.e. 'burnt'). In other words, a view-balance wallet would have the power to destroy enotes owned by the corresponding master wallet, which is sub-optimal (our thanks to Nikolas Krätzschmar for identifying this problem). The current linking tag construction prevents that issue by requiring knowledge of k_m in order to demonstrate the linking tag's discrete log.

2. Compute the nominal address tag: $\texttt{addr\_tag}' = \texttt{addr\_tag\_enc}$ XOR $\mathcal{H}_{ate}(s_1'^{sr}||K^o)$
3. Split the nominal address tag: $\texttt{addr\_tag}' \Rightarrow \{c'^j, \texttt{hint}'\}$
4. If $\mathcal{H}_{dch}(s_{ct}||c'^j) \neq \texttt{hint}'$ then ABORT.
5. Decipher the nominal address index: $j' = \texttt{Deciph}[s_{ct}](c'^j)$

With j' in hand, the user can recompute K^o. This requires $s_1'^{sr}$ and the user's s_{ga} and K_s.

1. Compute the address-index generator of j': $s_{gen}^{j'} = \mathcal{H}_{gen}(s_{ga}||j')$
2. Compute the spendkey extensions of j' for G, X, and U: $k_{[g/x/u]}^{j'} = \mathcal{H}_{se[g/x/u]}^n(K_s||j'||s_{gen}^{j'})$
3. Compute the address spend key of j': $K_1^{j'} = k_g^{j'}G + k_x^{j'}X + k_u^{j'}U + K_s$
4. Compute the nominal onetime address extensions: $k_{[g/x/u]}'^o = \mathcal{H}_{ko[g/x/u]}^n(K_1^{j'}||s_1'^{sr}||C)$
5. Compute the nominal onetime address: $K'^o = k_g'^o G + k_x'^o X + k_u'^o U + K_1^{j'}$
6. If $K'^o \neq K^o$ then ABORT.

Now the user is prepared to recover the enote amount.[24]

- **Normal scan**: This requires $s_1'^{sr}, j', s_{gen}^{j'}$, and the user's k_{ua} and K_s.
- **Selfsend scan**: This requires $s_1'^{sr}$ and the user's k_{vb}.

1. Compute the second enote secret:
   - **Normal scan**:
     (a) Compute the address private key of j': $k_a^{j'} = \mathcal{H}_{ap}^n(K_s||j'||s_{gen}^{j'})$
     (b) Compute the secret: $s_2'^{sr} = \mathcal{H}_{sr2n}((1/(k_a^{j'} * k_{ua})) * K_e)$
   - **Selfsend scan**: $s_2'^{sr} = \mathcal{H}_{sr2s}(k_{vb}||s_1'^{sr})$
2. Decrypt the nominal amount: $a' = a_{enc}$ XOR $\mathcal{H}_{ea}(s_1'^{sr}||s_2'^{sr})$
3. Compute the nominal amount blinding factor: $x' = \mathcal{H}_{bf}^n(s_1'^{sr}||s_2'^{sr})$
4. If $x'G + a'H \neq C$ then ABORT.

At this point the user is certain they own the enote. As a final step, they can compute the enote's key image. This requires $k_x^o, k_u^o, k_x^j, k_u^j$, and the user's k_{vb} and $k_m U$.

$$\tilde{K} = \frac{1}{k_x^o + k_x^j + k_{vb}} * ((k_u^o + k_u^j) * U + k_m U)$$

[24] These steps are not required for coinbase enotes, which record the amount a explicitly.

8.5.3 Implementing balance recovery

To actually perform balance recovery, users must examine enotes and key images in the ledger. We discuss an effective procedure for doing so.

First, we add a rule to Jamtis to ensure all transactions containing a user's key images will have at least one output enote that passes the view tag check with that user's k_{fr}.

- **Rule**: All transactions spending funds from a user's wallet must contain at least one selfsend enote.[25]

Now we define the balance recovery procedure.

1. Collect a chronologically contiguous range of transactions from the ledger. We assume the results of scanning all prior transactions have been cached by the user.
2. Perform view tag checks on those transactions and compute nominal address tags. Collect 'basic records' of enotes that pass the checks. A basic record includes the enote, its enote ephemeral public key, its input context, and its nominal address tag. We don't store K_d in basic records in case the source of those records is not trusted.

   Also collect the key images of transactions that contributed enotes to that basic record set.
3. Perform normal enote scanning on those basic records, starting with trying to decipher the nominal address tag. Then we recompute K_d and re-do the view tag check. After that, proceed normally.
4. Mark as spent all enotes owned by the user (including cached and newly acquired enotes) whose key images can be found in the transactions that fed enotes to the basic record set. Flag the transactions that contain key image matches.
5. Loop until there are no flagged transactions:
   (a) Perform full selfsend enote scanning on all basic records associated with the flagged transactions. Full selfsend scanning means testing all selfsend enote types against each basic record. We ignore the basic records' nominal address tags since selfsends have a separate derivation path for s^{sr}_1.
   (b) Mark as spent any selfsend enotes from the previous step whose key images can be found in the transactions associated with the basic record set. Flag those transactions (they may contain more selfsends).

With this procedure, we minimize the amount of work and data that needs to be handled by each successive step.

[25] To satisfy this rule, it may be necessary to add a dummy selfsend to a transaction if there is no change enote or explicit self-spend.

8.6 Jamtis wallets

The Jamtis key structure and balance recovery design lends itself to a well-defined set of 'wallet tiers'.

- **Find-received** (k_{fr}): Perform view tag checks for all enotes and compute nominal normal-enote address tags.
- **Generate-address** (s_{ga}, K_s): Generate any Jamtis address for arbitrary index j. Decipher the address index of any existing Jamtis address.
- **Payment validator** ($k_{fr}, s_{ga}, k_{ua}, K_s$): In addition to the find-received and generate-address tier capabilities, the payment validator can perform balance recovery for normal enotes up to (but not including) the key image computation.
- **View-balance** ($k_{vb}, k_m U$): View all balance-related information including for selfsend enotes.
- **Master** (k_{vb}, k_m): View the entire user's balance and make ownership/unspentness proofs for all enotes.

Here we see the value of k_{ua}, which separates payment validators from the find-received and generate-address wallet tiers.

8.6.1 Remote-assisted balance recovery

A user can set up a remote find-received wallet that creates basic records from enotes in the ledger and passes those records to their local view-balance wallet (or payment validator). There are several important details to highlight.

- A find-received wallet has very limited information about the user's owned enotes.
    1. It can perform view tag checks on all enotes, which essentially filters down the set of enotes 'possibly owned by the user'.
    2. It can uncover the nominal address tags of normal enotes, which means if the wallet knows about (or ever in the future learns about) addresses owned by the user, then it can identify all normal enotes owned by those addresses.
- A find-received wallet only has to transmit basic records of enotes that pass view tag checks and key images associated with those enotes to the user. This amounts to a substantial relative reduction in data that a user needs to access compared to the remote wallet, even if view tags are only 8 bits (a 256:1 reduction).
- The first step that a user needs to do to process a basic record is check the nominal address tag's hint, which is very cheap compared to the ECDH exchange required to obtain that hint. This means the user will spend very little computational time handling enotes they don't actually own, since hint checks will further filter out a significant proportion of unowned enotes (for a 16-bit hint it would be a 65536:1 reduction).

- The user will only look for selfsend enotes in transactions containing the key images of their owned enotes. Moreover, despite the selfsend path needing to decrypt the encrypted address tag before doing a hint check, doing so is very cheap since s_1^{sr} for selfsends is derived from k_{vb} instead of K_d. Overall this means checking for selfsends has very low relative computation cost.

The result is highly constrained information access for the remote find-received wallet and near-instantaneous balance recovery for the user.

8.7 Forward secrecy against DLP-solver

Jamtis is designed to offer conditional forward secrecy against a DLP-solver. This includes protecting amounts, the true signers in membership proofs, key image origins, and the identities of enote owners/recipients. Forward secrecy is conditional on the DLP solver not gaining access to a user's public addresses and their generate-address secret.[26]

8.7.1 Public address access

A DLP solver with a user's public Jamtis address can recover the user's find-received key and identify all normal enotes owned by that address and their amounts.

- The adversary can solve for k_{fr} in $k_{fr}K_3^j = K_2^j$ and use it to unwrap candidate onetime addresses to check if the unwrapped spend keys match the address's K_1^j.
- The adversary can solve for r in $rK_3^j = K_e$ to compute the key rG, which allows the s_2^{sr} of normal enotes to be computed.

If at least one of those normal enotes is spent, then the DLP solver can additionally compute all of the user's key images. More precisely, they can generate key image candidates derived from all on-chain key images, among which will be the real key images - if two normal enotes are spent then the candidate set collapses.

- The masked address decomposition of a spent Seraphis enote can be recovered by a DLP solver from the ownership/unspentness proof. This means the values $k_x^o + k_x^j + k_{vb}$ and $k_u^o + k_u^j + k_m$ of the original onetime address will be known to the adversary.

  Suppose the compromised user has spent at most one enote owned by each address known to the adversary. The adversary can use the decomposition values from all proofs in the ledger in combination with the k_x^o, k_u^o of that user's normal enotes to uncover nominal values $k_x^j + k_{vb}$ and $k_u^j + k_m$ for each spent enote in the ledger. Some of those pairs will correspond to the user's spent enotes. He can then combine those pairs with the k_x^o, k_u^o of other normal enotes owned by addresses known to him to compute sets of candidate key images, among which will be the real ones for all of the compromised enotes.

[26] Note that tevador, the author of Jamtis, has proposed embedding a post-quantum 'switch' into Seraphis that could be activated if a quantum adversary became credible [17].

However, if two enotes are spent out of the user's compromised enote set for a particular address, then the adversary will recover the same pair $k_x^j + k_{vb}$ and $k_u^j + k_m$ from both of those enotes, allowing the candidate set for that address to collapse.

A DLP solver cannot figure out anything about a user's selfsend owned enotes (aside from performing view tag checks if their find-received key is known). The DLP solver also cannot look at a set of addresses to learn anything about addresses not in that set.

8.7.2 Generate-address secret access

If the DLP solver gets access to a user's generate-address secret in addition to an address that owns spent enotes, then they can unravel the user's entire key structure (including the master key).

- This trivially follows from knowing $k_x^j + k_{vb}$ and $k_u^j + k_m$, as detailed in the previous section.

8.7.3 Unknown users

DLP solvers cannot learn anything about users whose addresses they don't know (aside from weakening their membership proofs by learning the true spends of compromised users). We support that claim with the following observations about Jamtis.

- Onetime addresses include sender extensions and address index extensions on all generators, ensuring no private keys are directly exposed to a DLP solver by ownership/unspentness proofs.
- Enote ephemeral public keys use a uniformly distributed value r, and the discrete log of K_e with respect to a public generator is never used.
- The Pedersen commitments used for amounts are perfectly hiding.
- The masked addresses and amount commitments in enote images are masked with uniformly distributed values t_k and t_c, ensuring the original enotes are perfectly hidden.
- Seraphis membership proofs are required to have [todo: relevant security properties].

8.8 Seraphis requirement satisfaction

Jamtis fully satisfies the information recovery requirements specified by Seraphis [9].

- **Secret recovery**: As detailed in Section 8.5.2, the values $k^o_{[g/x/u]}$, a, and x will be recovered during balance recovery.
- **Enote reproduction**: As detailed in Section 8.5.2, recomputing K^o and C is required to successfully complete balance recovery on an enote.

- **Enote privacy**: Since Jamtis enotes are constructed from ECDH secrets and the hidden private key k_{vb}, observers who know none of a user's Jamtis private keys cannot uncover any information about that user's enotes' contents. [todo: this requires a substantial proof]

9 Non-prime groups

This paper requires $\mathbb{G}$ to be a prime group, however in practice it may be a prime subgroup of a non-prime group. One prominent example, used in CryptoNote [20] and its progeny, is the elliptic curve Ed25519 [1], which has order $8 * l$ (l is a prime number $\approx 2^{252}$). CryptoNote enotes and proofs are designed to only use curve points from the subgroup of size l.

All uses of curve points in an implementation of Seraphis based on a non-prime group must take into account the possibility that a point recorded in a transaction may not be in the prime subgroup.

In particular, linking tags recorded in enote-images *must* be points in the prime subgroup [5], since checking if a linking tag has appeared in the ledger usually involves a byte-wise lookup. There are several ways to ensure non-prime points are detected by transaction validators. From least to most efficient, they are:

- Test $l * \tilde{K} \overset{?}{=} I$, where I is the group's identity element.
- Let the public key recorded in a transaction be $K_{precomputed} = (1/h) * K$, where h is the curve's cofactor (8 in the case of Ed25519). When validating a transaction, compute $K = h * K_{precomputed}$ to recover the point.
- Use an encoding abstraction such as Ristretto [3] that ensures all points recorded in a transaction (in enotes, enote-images, and proof elements) are in the prime subgroup.[27]

10 Modular transaction building

Seraphis, like other transaction protocols inspired by RingCT, does not include any advanced 'scripting' capabilities such as those found in Bitcoin. However, a Seraphis implementation can be designed to permit relatively more modular transaction building compared to RingCT and other protocols. A modular design enables membership proof deferment, membership proof delegation and transaction chaining.

- Membership proof deferment means delaying membership proofs to the very last step of transaction construction.[28]

[27] A Ristretto point will fail to decompress into a full elliptic curve point if it is not in the prime subgroup.

[28] Deferring membership proofs allows a transaction author to minimize timing information about when they constructed their transaction that might be leaked by membership proofs. This is especially advantageous for multisignature schemes where a transaction may take days or weeks to be constructed.

- Membership proof delegation means allowing a third party to construct an enote's membership proof.
- Transaction chaining is the ability to construct a transaction B that spends an enote produced by transaction A, before A has been added to the ledger.

Below is a transaction-building procedure that supports those techniques.

1. Define the transaction's output enotes and any miscellaneous memos.
2. Construct ownership/unspentness proofs for each input's enote-image. Each proof should sign a message containing all of the transaction's key images, output enotes, and memos, and the transaction fee. Cache the values t_c and t_k for each input.
3. Construct a balance proof for the transaction.
   - The individual who performs/completes a balance proof must know the blinding factors and amounts of all input enote-images and output enotes.
4. Construct range proofs for all of the output enotes' amount commitments {<u>and for the input enote-images' masked amount commitments</u>}.
5. Construct a membership proof for each input using the cached t_c and t_k values. Membership proofs should not sign any transaction material other than material directly related to the membership proof (e.g. the relevant enote-image and ledger references to the enotes referenced by the proof). Membership proofs and ownership/unspentness proofs should not share any Fiat-Shamir challenges.

With this procedure, membership proof deferment is trivially satisfied. Delegation is achieved by only needing the values $t_{c,j}$, $t_{k,j}$, C_j, and K^o_j to construct membership proofs. Transaction chaining is possible because the first four steps can be executed even if the inputs being spent don't exist in the ledger.

11 Other recommendations

The recommendations in previous sections are not exhaustive. Here are some other ideas we think implementers should consider.

- **Semantic constraints**: Transaction validation rules should contain as many 'semantic constraints' as possible. A semantic constraint is one that limits variance in how a transaction may be constructed, without affecting the underlying security model. For example, how inputs and outputs are sorted, byte serialization, memo field format/usage, etc.

  Reducing/eliminating semantic variance reduces the likelihood of 'implementation fingerprinting'. If two transaction-builder implementations use different semantic conventions, then observers can easily identify what software was used to make a given transaction. This can have undesirable privacy implications for users.

- **Decoy selection**: Membership proofs might have small reference sets relative to the ledger size. If 'decoy' enotes are not selected effectively, then observers may be able to use heuristics to gain an advantage when trying to guess the real spend in a transaction input.

  Pure random selection of decoys is weak to the 'guess-newest' heuristic, where the 'newest' enote referenced by a membership proof is most likely to be the real spend. Selecting from a gamma distribution instead is thought to better mimic the true spend distribution, and selecting 'bins' (clumps) of enotes mitigates analysis that uses circumstantial timing knowledge about a transaction. [13, 16]

- **Fee granularity**: If transaction fees have very high granularity then a lot of information about a transaction author may be inferred from the fees they use (e.g. their wallet implementation and when they constructed a transaction) [11]. This may be mitigated by 'discretizing' fee values, for example by only allowing powers of 1.5.

References

[1] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, Sep 2012. https://ed25519.cr.yp.to/ed25519-20110705.pdf [Online; accessed 03/04/2020].

[2] Heewon Chung, Kyoohyung Han, Chanyang Ju, Myungsun Kim, and Jae Hong Seo. Bulletproofs+: Shorter proofs for privacy-enhanced distributed ledger. Cryptology ePrint Archive, Report 2020/735, 2020. https://eprint.iacr.org/2020/735 [Online; accessed 07/17/2021].

[3] Henry de Valance, Isis Lovecruft, and Tony Arcieri. Ristretto. https://ristretto.group/ristretto.html [Online; accessed 10/05/2020].

[4] Justin Ehrenhofer and knacc. Advisory note for users making use of subaddresses, October 2019. https://web.getmonero.org/2019/10/18/subaddress-janus.html [Online; accessed 01/02/2020].

[5] Riccardo "fluffypony" Spagni and luigi1111. Disclosure of a Major Bug in Cryptonote Based Currencies, May 2017. https://getmonero.org/2017/05/17/disclosure-of-a-major-bug-in-cryptonote-based-currencies.html [Online; accessed 04/10/2018].

[6] Brandon Goodell, Sarang Noether, and RandomRun. Concise Linkable Ring Signatures and Forgery Against Adversarial Keys. Cryptology ePrint Archive, Report 2019/654, 2019. https://eprint.iacr.org/2019/654 [Online; accessed 11/23/2020].

[7] Aram Jivanyan and Aaron Feickert. Lelantus spark: Secure and flexible private transactions. Cryptology ePrint Archive, Report 2021/1173, 2021. https://ia.cr/2021/1173 [Online; accessed 09/19/2021].

[8] kayabaNerve. Remove the burning bug as a class of attack with a modified shared key definition, Issue #103, May 2022. https://github.com/monero-project/research-lab/issues/103 [Online; accessed 04/21/2023].

[9] koe. Seraphis draft (WIP), April 2023. https://github.com/UkoeHB/Seraphis [Online; accessed 04/05/2023].

[10] koe, Kurt M. Alonso, and Sarang Noether. Zero to Monero — Second Edition, April 2020. https://web.getmonero.org/library/Zero-to-Monero-2-0-0.pdf [Online; accessed 10/03/2020].

[11] Mitchell Krawiec-Thayer. MoneroKon 2019 - Visualizing Monero: A Figure is Worth a Thousand Logs, June 2019. https://www.youtube.com/watch?v=XIrqyxU3k5Q [Online; accessed 01/06/2020].

[12] Paul J. Leach, Rich Salz, and Michael H. Mealling. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122, July 2005.

[13] Andrew Miller, Malte Möser, Kevin Lee, and Arvind Narayanan. An Empirical Analysis of Linkability in the Monero Blockchain. *CoRR*, abs/1704.04299, 2017. https://arxiv.org/pdf/1704.04299.pdf [Online; accessed 03/04/2020.

[14] Sarang Noether and Brandon Goodell. An efficient implementation of Monero subaddresses, MRL-0006, October 2017. https://web.getmonero.org/resources/research-lab/pubs/MRL-0006.pdf [Online; accessed 04/04/2018].

[15] Sarang Noether and Brandon Goodell. Triptych: logarithmic-sized linkable ring signatures with applications. Cryptology ePrint Archive, Report 2020/018, 2020. https://eprint.iacr.org/2020/018 [Online; accessed 03/04/2020.

[16] Viktoria Ronge, Christoph Egger, Russell W. F. Lai, Dominique Schröder, and Hoover H. F. Yin. Foundations of ring sampling. Cryptology ePrint Archive, Report 2020/1550, 2020. https://ia.cr/2020/1550 [Online; accessed 09/19/2021].

[17] tevador. [Draft] Zero-cost post-quantum mitigations for Seraphis, September 2022. https://gist.github.com/tevador/23a84444df2419dd658cba804bf57f1a [Online; accessed 04/21/2023].

[18] tevador. JAMTIS, December 2023. https://gist.github.com/tevador/50160d160d24cfc6c52ae02eb3d17024 [Online; accessed 04/05/2023].

[19] UkoeHB. Reduce scan times with 1-byte-per-output 'view tag', Issue #73, April 2020. https://github.com/monero-project/research-lab/issues/73 [Online; accessed 11/23/2020].

[20] Nicolas van Saberhagen. CryptoNote V2.0. https://bytecoin.org/old/whitepaper.pdf [Online; accessed 03/10/2021].

A Jamtis information proofs

Jamtis is designed so users may generate proofs about various pieces of information related to their accounts.

In this Appendix we will use Seraphis composition proofs [9] to prove knowledge of the composition of public keys that have the construction $K = aG + bX + cU$, and to prove that key images $\tilde{K}$ are derived from those K. Constructing a composition proof requires a, b, c and a message m to sign, and produces a proof σ_{cp} and key image $\tilde{K} = (b/c) * U$. Verifying the proof requires $\sigma_{cp}, K, \tilde{K}$ and m. [todo: abstract away the composition proof and just use 'any proof that satisfies the Seraphis ownership/unspentness requirements'?]

A.1 Address ownership proof

An address ownership proof proves that a Jamtis address K is owned by the prover (or that the prover is cooperating with the owner, which is equivalent). This proof must be a Σ-protocol, which in practice means binding to a custom message m_{custom} provided by the verifier. Without that binding, a proof could be reused by someone who is not the address owner.

In this proof, K may equal the Jamtis base spend key $K_s = k_{vb}X + k_m U$ or a full Jamtis address key $K_1^j = k_g^j G + (k_x^j + k_{vb})X + (k_u^j + k_m)U$.

Since K might have no G component, we include an offset on G in the proof.

Prover

1. Compute proof offset $k_{g,o} = \mathcal{H}_{ao}^n(K)$.
2. Create σ_{cp} and $\tilde{K}$ from $K_o = k_{g,o}G + K$ and m_{custom}.

An address ownership proof is the tuple $\Omega_{ao} = [m_{custom}, K, \tilde{K}, \sigma_{cp}]$.

Verifier

Given a proof Ω_{ao}:

1. Check that m_{custom} and K match the expected values.
2. Compute proof offset $k_{g,o} = \mathcal{H}_{ao}^n(K)$.
3. Verify σ_{cp} using $K_o = k_{g,o}G + K$ and the provided $\tilde{K}, m_{custom}$.

A.2 Address index proof

An address index proof proves that a Jamtis address key K_1^j was constructed from an index j.

Prover

Construct the proof tuple $\Omega_{ai} = [K_s, j, s^j_{gen}, K_1^j]$.

Verifier

Given a proof Ω_{ai}:

1. Check that K_1^j matches the expected value.
2. Compute the spendkey extensions of j for G, X, and U: $k^j_{[g/x/u]} = \mathcal{H}^n_{se[g/x/u]}(K_s||j||s^j_{gen})$
3. Compute the nominal address spend key: $K_1'^j = k^j_g G + k^j_x X + k^j_u U + K_s$
4. Verify that $K_1'^j \overset{?}{=} K_1^j$.

Jamtis design comment

Here we see that s^j_{gen} hides s_{ga} from the verifier, and the fact it is bound to j prevents the verifier from using it to create other addresses. We also see that explicitly binding $k^j_{[g/x/u]}$ to j allows address index proofs to demonstrate that K_1^j has a relationship with j.

Binding $k^j_{[g/x/u]}$ to K_s is required so you cannot make an address index proof for an arbitrary j' on any arbitrary key $K_1^{j"}$ [todo: formalize this security property]. Binding to K_s also allows a user to combine address index proofs for many j with a single address ownership proof on K_s. This could allow a generate-address Jamtis wallet to create address index proofs for arbitrary indices and combine them with an address ownership proof on K_s provided by the master wallet.

A.3 Enote ownership proof

An enote ownership proof proves that an enote with $[K^o, C]$ is owned by a key K. It can be combined with an address ownership proof on K to demonstrate that the prover can spend the enote.

An enote ownership proof may be constructed by the enote's author or its owner.

Prover

1. Compute the first sender-receiver secret using the appropriate derivation path for K: s^{sr}_1.

The proof is $\Omega_{eo} = [K, K^o, C, s^{sr}_1]$.

Verifier

Given a proof Ω_{eo}:

1. Check that K^o and C match the expected values.
2. Compute the onetime address extensions: $k^o_{[g/x/u]} = \mathcal{H}^n_{ko[g/x/u]}(K||s^{sr}_1||C)$
3. Compute the nominal onetime address: $K'^o = k^o_g G + k^o_x X + k^o_u U + K$
4. Check that $K'^o \stackrel{?}{=} K^o$.

Jamtis design comment

Here we see that binding $k^o_{[g/x/u]}$ to K allows enote ownership proofs to demonstrate a relationship between K and K^o.

Note that this proof is agnostic to the derivation path for s^{sr}_1, meaning the verifier has no way to know the enote's type (whether it's normal or a selfsend type). He also does not know if the proof was made by the enote's author or its owner.

A.4 Enote amount proof

An enote amount proof proves that an amount commitment $C = xG + aH$ binds an amount a. The proof is simply $\Omega_{ea} = [C, x, a]$, which is verified by testing $xG + aH \stackrel{?}{=} C$.

A.5 Enote key image proof

An enote key image proof proves that a key image $\tilde{K}$ is derived from a onetime address K^o.

Prover

1. Create a composition proof on K^o: σ_{cp} with a null message m_0.

The proof tuple is $\Omega_{eki} = [K^o, \tilde{K}, \sigma_{cp}]$.

Verifier

Given a proof Ω_{eki}:

1. Check that K^o and $\tilde{K}$ match the expected values.
2. Verify that $\tilde{K}$ is a canonical group element.
3. Verify σ_{cp} with the given K^o and $\tilde{K}$ and null message m_0.

A.6 Enote unspent proof

An enote unspent proof proves that a key image $\tilde{K}$ was *not* derived from a onetime address K^o. This can be used to show that transactions in the ledger did not spend a particular enote.[29]

For this proof we need a matrix proof θ (see Appendix B).

Prover

Given a test key image $\tilde{K}'$ and a onetime address $K^o = k_g G + k_x X + k_u U$ where k_g, k_x, k_u are known to the prover:

1. Compute the following:
$$\begin{aligned} K_g &= k_g G \\ K_x &= k_x X \\ K_u &= k_u U \\ \tilde{K}'_x &= k_x \tilde{K}' \end{aligned}$$
2. Create a matrix proof demonstrating the discrete log k_g of K_g with respect to G: θ_g.
3. Create a matrix proof demonstrating the discrete log k_x of K_x and $\tilde{K}'_x$ with respect to X and $\tilde{K}'$: θ_x.
4. Create a matrix proof demonstrating the discrete log k_u of K_u with respect to U: θ_u.

The proof tuple is $\Omega_{eu} = [K^o, \tilde{K}', \tilde{K}'_x, \theta_g, \theta_x, \theta_u]$.

Verifier

Given a proof Ω_{eu}:

1. Check that K^o and $\tilde{K}'$ match the expected values.
2. Check that $K^o \stackrel{?}{=} K_g + K_x + K_u$.
3. Verify the matrix proofs $\theta_g, \theta_x, \theta_u$.
4. If $\tilde{K}'_x \stackrel{?}{=} K_u$ then $\tilde{K}'$ is the key image of K^o.[30]

[29] Credit for the design of this proof belongs to pseudonymous Monero contributor dangerousfreedom.

[30] The verifier must ensure that $\tilde{K}'_x$ and K_u are canonical group elements.

A.7 Transaction funded proof

A transaction funded proof proves that the prover owns the enote that has key image $\tilde{K}$. This proof must be a Σ-protocol, which in practice means binding to a custom message m_{custom} provided by the verifier. Without that binding, a proof could be reused by someone who does not own the proof enote.

A transaction funded proof can be used to show that the prover contributed funds to a transaction. Importantly, the enote used in the proof is not exposed to the verifier. This means the amount in the enote is also not exposed. The prover can expose that amount without exposing the original enote by making an enote amount proof on the masked commitment of the relevant transaction input (assuming they cached the blinding factor t_c, otherwise they would need to make an enote amount proof on the original enote's amount commitment).

Prover

Given a key image $\tilde{K}$ of a onetime address K^o owned by the prover, and a message m_{custom}:

1. Generate a random mask $t_k \in_R \mathbb{Z}_l$ and compute $K' = t_k G + K^o$.
2. Make a composition proof on K': σ_{cp}.

The proof tuple is $\Omega_{tf} = [m_{custom}, K', \tilde{K}, \sigma_{cp}]$.

Verifier

Given a proof Ω_{tf}:

1. Check that m_{custom} and $\tilde{K}$ match the expected values.
2. Verify σ_{cp} on the given $m_{custom}, K', \tilde{K}$.

A.8 Enote sent proof

An enote sent proof proves that an enote with amount a and onetime address K^o was sent to an address K.

This is trivially achieved with an enote ownership proof Ω_{eo} and enote amount proof Ω_{ea}. The verifier should expect that the two proofs use values of K^o and C that come from the same enote.

A.9 Reserved enote proof

A reserved enote proof proves that an enote with onetime address K^o is owned by an address K, has amount a, has key image $\tilde{K}$, is onchain, and is unspent.

Prover

Given an enote owned by the prover:

1. Create an enote ownership proof: Ω_{eo}.
2. Create an enote amount proof: Ω_{ea}.
3. Create an enote key image proof: Ω_{eki}.
4. Identify the enote's index in the ledger: i.

The proof tuple is $\Omega_{re} = [\Omega_{eo}, \Omega_{ea}, \Omega_{eki}, i]$.

Verifier

Given a proof Ω_{re}:

1. Look up the enote values $[K^o, C]$ in the ledger using i. Expect those values match with each of the sub-proofs.
2. Check that i matches the expected value (or, conversely, that the recovered values $[K^o, C]$ match their expected values).
3. Check the sub-proofs $\Omega_{eo}, \Omega_{ea}, \Omega_{eki}$.
4. Check that the key image $\tilde{K}$ in Ω_{eki} does not exist in the ledger.

A.10 Reserve proof

A reserve proof proves that the prover has at least $v = \sum a$ unspent funds in the ledger. This proof must be a Σ-protocol, which in practice means binding to a custom message m_{custom} provided by the verifier. Without that binding, a proof could be reused by someone who does not own the reserved funds.

Prover

Given a set of n enotes and the m Jamtis addresses that own them, and a custom message m_{custom}:

1. Randomize the order of the proof enotes.
2. Make reserved enote proofs for all of the enotes: $\{\Omega_{re}\}_n$.
3. Make address ownership proofs for all of the addresses that own enotes in the reserve proof: $\{\Omega_{ao}\}_m$. Use the custom message m_{custom}.

The proof tuple is $\Omega_{rp} = [\{\Omega_{re}\}_n, \{\Omega_{ao}\}_m, m_{custom}]$.

Verifier

Given a proof Ω_{rp}:

1. Check that m_{custom} matches the expected value.
2. Verify the address ownership proofs $\{\Omega_{ao}\}_m$ using the message m_{custom}.
3. Verify the reserved enote proofs $\{\Omega_{re}\}_n$ using the addresses found in $\{\Omega_{ao}\}_m$.
4. Verify that the enotes in $\{\Omega_{re}\}_n$ are in the ledger and unspent.
   (a) Use the ledger indices i to obtain copies of enotes from the ledger, then check that their values match with $\{\Omega_{re}\}_n$.
   (b) Look up each reserved enote's key image $\tilde{K}$ in the ledger to check if the enotes are unspent as expected (the key images can be found in the enote key image sub-proofs).

The verifier can sum together amounts a stored in the enote amount sub-proofs in $\{\Omega_{re}\}_n$ in order to get the total amount v that is reserved by the prover.

Note that a valid reserve proof may become invalid immediately after verification, since the enotes referenced in the proof may be spent at any time.

B Matrix proof

We present an extended Schnorr-like proof for proving knowledge of discrete logs in a two-dimensional matrix between a vector of private keys and a vector of base keys. The powers-of-μ aggregation technique used here is inspired by [15].

B.1 Construction

Our proof is a Schnorr-like Σ-protocol between prover and verifier.

Suppose the prover has keys k_i for $i \in 0, ..., N-1$, and is given base keys B_j for $j \in 0, ..., M-1$. Let him compute the public key matrix $\mathbb{M}_{i,j} = k_i$ x B_j for all i, j.

1. The prover generates random scalar $\alpha \in_R \mathbb{Z}_l$.
2. The prover computes $A_j = \alpha B_j$ for all j. He sends all B_j, A_j, and $\mathbb{M}$ to the verifier.
3. The verifier generates random aggregation factor $\mu \in_R \mathbb{Z}_l$ and random challenge $c \in_R \mathbb{Z}_l$ and sends them to the prover.
4. The prover computes aggregate response r and sends it to the verifier (we use exponents of μ here).

$$r = \alpha - c * \sum_{i=0}^{N-1} \mu^i * k_i$$

5. The verifier computes

$$A'_j = rB_j + c * \sum_{i=0}^{N-1} \mu^i * \mathbb{M}[i][j]$$

for all j and checks $A'_j \overset{?}{=} A_j$.