Salvium review

Cypher Stack*

July 19, 2024

This report contains a review of an informal description of the Salvium protocol. As with any such report, it may contain errors and cannot guarantee correctness or security. Further, it cannot guarantee that any particular implementation of the construction is correct, secure, or suitable for intended use cases.

The author asserts no warranty and disclaims liability for its use. The author further expresses no endorsement of any kind. This report has not undergone any further formal or peer review.

Contents

*https://cypherstack.com

1 Introduction

Salvium is a transaction protocol built as a modification of the Monero protocol, with an associated forked codebase.

One feature it seeks to add is return functionality, in which the sender of a transaction safely includes data that the recipient can use to generate a later transaction sending funds back to the original sender.

Separately, the protocol adds more general transaction types. These types may require that transaction amounts no longer strictly balance. Normally, consensus rules require that the sum of consumed output values precisely equal the sum of generated output values (along with a plaintext fee) without revealing any of the output values to the network.

Salvium provided informal descriptions of the design aspects for these two goals to Cypher Stack. In this report, we expand on this descriptions for the purposes of improved clarity and analysis, present them using more standard notation, and determine the extent to which the protocol is likely to achieve the goals. We caution, however, that a formal security model of the transaction protocol was not provided; as such, any such analysis is inherently limited. Further, this analysis does not reflect any particular implementation of the protocol, and cannot make any conclusions about such implementations.

Throughout the report, we generally use notation from the "Zero to Monero" documentation[1] by Monero contributor `koe`, as its notation is thorough, very consistent, and as such an increasingly *de facto* standard for Monero protocol research and development.

2 Return functionality

For the sake of clarity, suppose that Alice wishes to generate a transaction sending funds to Bob.

Salvium seeks to provide return functionality, whereby Alice can securely communicate in-band data to Bob that enables him to later generate a transaction sending funds back to Alice.

Informally, any design meeting this goal should have the following properties:

1. A transaction containing return data must not be distinguishable from a transaction not containing return data to an observer (aside from unrelated metadata).

2. Multiple transactions containing return data for Alice must not be linkable to an observer (aside from unrelated metadata).

3. If Alice does not wish to include return data, Bob must be able to detect this.

[1] `https://github.com/UkoeHB/Monero-RCT-report`

4. If Alice includes return data, it must be reusable; Bob must be able to generate multiple return transactions using the data, all of which Alice can properly receive.

5. Any entity aside from Bob (or his delegate) must not be able to recover the return data, if any, from a transaction.

6. Any entity aside from Alice (or her delegate) must not be able to receive any return transaction.

Because Salvium transactions (like those in Monero) permit storage of arbitrary associated data, a simple approach to this is to encrypt Alice's address to Bob and include it as additional data. On receipt of the transaction, Bob can decrypt the address and use it for a later return transaction that directs funds to Alice. However, this is somewhat inefficient. Salvium addresses use a dual-key design that consists of two elliptic curve group elements, each of which encodes to 32 bytes. This implies (at least) an extra 64 bytes of data per transaction.

Instead, Salvium's design for return functionality follows a proposal[2] by Monero contributor `knaccc` that is more efficient. We describe this design here with updated notation for completeness. Note that while the linked design supports transactions sending to multiple recipients, Salvium restricts this to transactions sending only to a single recipient.

2.1 Protocol for return functionality

Suppose that Alice has a secret keypair (k_A^v, k_A^s) and corresponding address (K_A^v, K_A^s), and that Bob has a secret keypair (k_B^v, k_B^s) and corresponding address (K_B^v, K_B^s), where each address component is computed by multiplication of the corresponding secret key component by group generator G. It may be the case that either of Alice's or Bob's addresses is a subaddress. Alice wishes to produce a transaction sending funds to Bob, such that Bob should be able to extract data sufficient to later produce a return transaction to Alice.

Alice constructs her transaction to Bob mostly as usual, so we describe only the steps relevant for the updated design. When constructing her transaction, Alice generates a one-time output key K^o to Bob:

1. Samples a scalar r uniformly at random.

2. Computes the one-time output key $K^o = \mathsf{H}_n(rK_B^v)G + K_B^s$.

3. If Bob's address is a subaddress, computes $R = rK_B^s$ as the transaction key; otherwise, computes $R = rG$ as the transaction key.

4. Continues generating the output as usual, including K^o and R in the transaction.

[2] `https://github.com/monero-project/research-lab/issues/53`

She also generates a one-time change output

$$K_c^o = \mathsf{H}_n(k_A^v R)G + K_A^s$$

to herself as usual. Note that this procedure uses the transaction key R from above; this is well defined since Salvium permits only transactions consisting of a single non-change output. Then, Alice computes the value

$$y = \mathsf{H}_n\left(\texttt{"refund"}, \mathsf{H}_n(rK_B^v)\right)$$

and includes the value $F = y^{-1}k_A^v K_c^o$ in the transaction as additional data.

Note that the only modification from a standard transaction is the inclusion of the value F, which does not require a change to consensus rules.

Bob receives the funds as usual, but also wishes to recover a return subaddress that he can use to later return funds to Alice. If Bob's address is a subaddress, he computes

$$\begin{aligned} y &= \mathsf{H}_n\left(\texttt{"refund"}, \mathsf{H}_n(k_B^v R)\right) \\ &= \mathsf{H}_n\left(\texttt{"refund"}, \mathsf{H}_n(k_B^v(rK_B^s))\right) \\ &= \mathsf{H}_n\left(\texttt{"refund"}, \mathsf{H}_n(r(k_B^v K_B^s))\right) \\ &= \mathsf{H}_n\left(\texttt{"refund"}, \mathsf{H}_n(rK_B^v)\right) \end{aligned}$$

to recover y. Otherwise, the same computation yields

$$\begin{aligned} y &= \mathsf{H}_n\left(\texttt{"refund"}, \mathsf{H}_n(k_B^v R)\right) \\ y &= \mathsf{H}_n\left(\texttt{"refund"}, \mathsf{H}_n(k_B^v(rG))\right) \\ &= \mathsf{H}_n\left(\texttt{"refund"}, \mathsf{H}_n(r(k_B^v G))\right) \\ &= \mathsf{H}_n\left(\texttt{"refund"}, \mathsf{H}_n(rK_B^v)\right) \end{aligned}$$

to recover y. In both cases, this is the same value generated by Alice. Then, he computes $yF = k_A^v K_c^o$. At this point, Bob defines

$$(K_A^{v,r}, K_A^{s,r}) = (yF, K_c^o) = (k_A^v K_c^o, K_c^o)$$

as Alice's return subaddress.

Bob can then construct return transactions to Alice as usual using her return subaddress. When he does so, he generates a one-time output key $K^{o,r}$ to Alice:

1. Samples a scalar r' uniformly at random.
2. Computes the one-time output key $K^{o,r} = \mathsf{H}_n(r'K_A^{v,r})G + K_A^{s,r}$.
3. Computes $R' = r'K_A^{s,r}$ as the transaction key.
4. Continues generating the output as usual.

To show that Alice can receive funds to her return subaddress, observe that she computes the following as part of the usual process:

$$\begin{aligned} K^{o,r} - \mathsf{H}_n\left(k_A^v(r'K_A^{s,r})\right)G &= \mathsf{H}_n(r'K_A^{v,r})G + K_A^{s,r} - \mathsf{H}_n\left(r'(k_A^v K_c^o)\right)G \\ &= \mathsf{H}_n(r'K_A^{v,r})G + K_A^{s,r} - \mathsf{H}_n(r'K_A^{v,r})G \\ &= K_A^{s,r} \end{aligned}$$

Provided that Alice has previously stored $K_A^{s,r} = K_c^o$ in her subaddress lookup table, she will associate the funds with her original transaction to Bob.

2.2 Protocol for non-return functionality

It may be the case that Alice does not wish to include return data in her transaction to Bob. If she simply did not include the value F as associated data, any network observer could trivially distinguish this case, which violates the design requirements. It is not, however, sufficient to include a random value for F, since Bob would be unable to identify that the corresponding reconstructed return subaddress was not intended to be used, and any funds sent to the subaddress could not be received by Alice and would effectively be lost.

Instead, Alice computes y as above, but sets $F = y^{-1}G$. If Bob subsequently computes $yF = G$ during the subaddress recovery process, he identifies that Alice has not provided return data and aborts.

Salvium notes that they do not provide for this functionality in their implementation. However, it is described in the original proposal, so we describe it here for completeness. It is also not possible for the network to detect if this functionality is used, as it is semantically identical to the use of return functionality.

2.3 Analysis

2.3.1 Design requirements

We must show that this construction meets the informal design requirements given above.

Requirement 1 is that a transaction not including return data be indistinguishable to an observer from a transaction not including this data. To see why this should hold, observe first that a transaction not including return data supplies the value $F = y^{-1}G$. If the nonce r is sampled uniformly at random and H_n is a secure cryptographic hash function with output uniformly distributed in $\mathbb{Z}_l$, then y is also distributed uniformly at random. This implies that F is as well. (Note that the case $y = 0$ occurs only with negligible probability, in which case F is undefined and Alice resamples the nonce.) In the case where a transaction includes return data, we have $F = y^{-1}k_A^v K_c^o$ instead. Provided that $k_A^v K_c^o \neq 0$, this group element is a generator, and the same reasoning applies.

Requirement 2 is that multiple transactions including return data for Alice should not be linkable by an observer. This follows immediately from the same reasoning as above.

Requirement 3 is that Bob be able to determine if Alice included return data or not. As noted, if Bob computes $yF = G$, he assumes that Alice did not wish to include return data. This process therefore fails only if there is a collision $k_A^v K_c^o = G$, which occurs if Alice is honest only with negligible probability.

Requirement 4 is that return data be reusable, such that Bob can use it to produce as many return transactions as it wishes. This follows immediately since the return data recovered by the recipient is constructed as a semantically-valid subaddress.

Requirement 5 is that no entity aside from Bob (or his delegate) can recover return data from a transaction. Given transaction data that includes F, it must be infeasible for such an entity to produce yF. This almost certainly holds given the construction of y, which is the output of a domain-separated cryptographic hash function whose input is itself a hash-based derivation of a sender-recipient shared secret. This shared secret can be computed given the recipient's key k_B^v, which can be delegated for scanning purposes. Formalizing this reasoning given all other transaction data requires a more complex security model.

Requirement 6 is that no entity aside from Alice (or her delegate) can receive funds sent in a return transaction. This effectively requires only that no component of the secret keypair corresponding to a return subaddress be computable by an adversary. Given that the secret key k_c^o corresponding to the change output one-time key $K_c^o = k_c^o G$ is computable only by Alice (as is the case under the Monero protocol's implicit security model), and given that the return subaddress is valid, this follows immediately.

2.3.2 Subaddress implications

The return address that Bob computes for Alice has the form of a subaddress. As noted, this means Alice must use a lookup table during the scanning process in order to receive the return transaction from Bob. Specifically, Alice computes the value

$$K_A^{s,r} = K_c^o = \mathsf{H}_n(k_A^v R)G + K_A^s$$

from this process. She then must query a local table to map this value onto data sufficient for her to recover the subaddress index. When she later wishes to spend the funds, she must further recover k_c^o.

For general subaddress-destined transaction receipt, it suffices for this table to map a subaddress component K_A^s to the index used to generate it; this allows the sender to compute k_A^s, which is required to then compute the one-time output signing key. In the case of a return subaddress, this is made more complex since the index corresponding to the subaddress component K_A^s used to produce the change output K_c^o is not sufficient to produce k_A^s (and then k_c^o), as Alice also requires access to the transaction key R from the transaction that she originally used to produce the change output (in addition to the transaction R' from the return transaction sent from Bob).

Salvium must therefore ensure that the subaddress lookup table design used for this purpose accounts for the additional data required for Alice to identify

the subaddress index associated to a return subaddress, as well as to compute k_c^o in order to later spend the funds.

The use of subaddresses also means that Alice may be vulnerable to the Janus attack, where an attacker attempts to link potential subaddresses. To execute the attack in the context of return transactions, Bob receives two transactions from Alice, and uses return subaddress components from both transactions to produce a "hybrid" subaddress to which he produces a return transaction. If Alice indicates acceptance of this transaction (either out of band or using other heuristics or analysis), then Bob knows they are linked.

We note that this attack is not unique to the return address context, and applies to subaddresses generally. However, any transaction with a return address generates a new subaddress and is therefore susceptible with respect to any other such transaction; this differs from the more general case where new subaddresses are generated only on demand.

2.3.3 Return data encryption

We note a modification that may be useful for analysis or efficiency. Specifically, the value F may be computed in a simpler manner that does not require scalar or group operations.

Instead of computing

$$y = \mathsf{H}_n\left(\texttt{"refund"}, \mathsf{H}_n(rK_B^v)\right)$$

and then $F = y^{-1}k_A^v K_c^o$ when producing a return transaction, Alice instead produces a symmetric encryption key using a key derivation function on input $\mathsf{H}_n(rK_B^v)$, and then encrypts $k_A^v K_c^o$ to obtain F using this key. If Alice does not wish to include return data, she instead encrypts any agreed-upon 32-byte invalid point encoding (such as all zero bytes) to signal this.

Instead of computing (yF, K_c^o) as the return subaddress, Bob instead performs the same key derivation, decrypts F using this key to obtain $k_A^v K_c^o$, and then uses $(k_A^v K_c^o, K_c^o)$ as the return subaddress. If this decryption yields the distinguished invalid encoding, Bob knows that Alice did not wish to include return data, and aborts.

As the Salvium protocol already uses a simple XOR operation for encryption of data against a keystream, this approach could be used here; however, it is crucial that the key derivation be performed securely to avoid attacks. A safer (though less efficient) design would use a more standard stream cipher construction, such as `ChaCha12`; this is more flexible for plaintext length, and a fixed initialization vector may be used since the derived key is always unique.

3 Transaction types

Aside from standard transfer transactions, Salvium introduces four additional transaction types: three that may be constructed by users, and one that is constructed by block producers.

3.1 Transfer transactions

In transfer transactions, consumed and generated output values must balance (accounting also for a transaction fee). To protect values from being known by an adversary, non-fee values are bound computationally using Pedersen commitments.

Suppose a transaction consumes m outputs, each represented by a Pedersen commitment

$$C_j^a = x_j G + a_j H$$

for $1 \leq j \leq m$; here x_j is the mask and a_j the value. Suppose also that the transaction generates p outputs, each similarly represented by a Pedersen commitment

$$C_t^b = y_t G + b_t H$$

for $0 \leq t < p$. Assume the transaction also has an associated fee f. The transaction balances if and only if

$$\sum_{j=1}^{m} a_j - \left(\sum_{t=0}^{p-1} b_t + f \right) = 0$$

holds.

The transaction contains, for each consumed output, a corresponding so-called pseudo-output commitment of the form

$$C'^a_j = x'_j G + a_j H$$

to the same amount, but with a different mask. Masks are also constructed such that

$$\sum_{j=1}^{m} x'_j - \sum_{t=0}^{p-1} y_j = 0$$

holds. Using this, it is the case (except with negligible probability) that

$$\sum_{j=1}^{m} C'^a_j - \left(\sum_{t=0}^{p-1} C_t^b + fH \right) = 0 \tag{1}$$

if and only if the transaction balances; this equation is checked by verifiers for each transaction.

Because commitments are elements of a finite cyclic group, it is important that they not bind to values that can result in overflow during the evaluation of Equation 1. To ensure this cannot occur, each generated output commitment comes equipped with a range proof asserting that the represented value is contained within a range $[0, 2^{64})$ whose upper bound is much (much!) smaller than the group order (approximately 2^{252}). The fee f is limited to this range as well; because it is presented in the clear, there is no need for an associated range proof.

3.2 Additional transaction types

In addition to these transfer transactions, Salvium introduces the following transaction types (available to users) where the sum of consumed output values strictly exceeds the sum of generated output values (including the fee):

- *Burn*. In this transaction type, excess value is discarded; this effectively deflates the available supply.
- *Convert*. In this transaction type, excess value is used to mint a new output of a different asset type.
- *Yield*. In this transaction type, excess value is used to compute a later payout amount.

In each of these cases, the excess value v is included as plaintext additional transaction data. This requires that Equation 1 be modified in a straightforward way:

$$\sum_{j=1}^{m} {C'}_j^a - \left(\sum_{t=0}^{p-1} C_t^b + fH \right) = vH$$

Verifiers must check that the excess value v is contained within the range $[0, 2^{64})$.

Salvium notes that while their implementation is able to support convert transactions, they are currently disabled and therefore unavailable to users.

There is an additional transaction type not available to users, and which is generated by block producers on each block:

- *Protocol*. This transaction type performs two tasks:
  - For each convert transaction in the block, it mints a new output of a specified asset type; the value of this output is determined by an oracle query.
  - For each yield transaction in the block occurring 21600 blocks prior, it mints a new output whose value is determined by the yield transaction's excess value according to an informally-described formula.

3.3 Destination specification

For convert and yield transactions, it is necessary for the intended destination of newly-minted outputs to be specified in a manner that can be checked by verifiers. That is, a verifier must be able to inspect the corresponding protocol transaction and assert that the destination for each of its minted outputs is correct. Otherwise, a block producer could substitute a valid minted output for one of its own choosing with an unintended destination in a manner not detectable by the network.

Simply using the proposed return functionality is insufficient, since its design requirements assert that no observer be able to determine the return subaddress (if any) associated with a transaction; further, the existing informal security

model does not permit such an observer to identify the address associated with any honestly-generated transaction.

Salvium addresses this issue by modifying the structure of convert and yield transactions. In such transactions, the sender follows the steps for return functionality. However, instead of including the value F in the transaction as additional data, it instead does the following:

1. Samples a scalar r' uniformly at random.
2. Computes $R' = rK_A^s$.
3. Sets $(K_A^{v,r}, K_A^{v,s}) = (k_A^v K_c^o, K_c^o)$ as in the return protocol.
4. Computes $K^{o,r} = \mathsf{H}_n(k_A^v R')G + K_A^{s,r}$.
5. Includes the values R' and $K^{o,r}$ in the transaction as additional data.

Effectively, the sender plays the role of a recipient generating a return transaction, where R' is computed as a transaction key and $K^{o,r}$ as a one-time output key.

When generating a protocol transaction, a block producer respectively uses R' and $K^{o,r}$ as the transaction and one-time output keys when minting new outputs. The presence of these values in each original convert and yield transaction means the network can verify such outputs are correctly generated. Further, because the construction of these values follows the return transaction protocol, the sender can recover protocol transaction outputs.

We note an important benefit to this design: that the construction does not leak the sender's address (whether a subaddress or otherwise) to block producers or the network, as the output is constructed in the usual manner. Further, we observe that neither block producers nor the network can determine which output in a convert or yield transaction is a change output. To do so would require such an entity to compute the Diffie-Hellman shared secret used to produce $K^{o,r}$, which is infeasible without knowledge of the secret value k_A^v.

3.4 Analysis

3.4.1 Censorship risk

Transfer transactions reduce the amount of plaintext data available to the network. For instance, they protect output values using commitments. This is useful in part because it reduces the opportuinity for network participants and block producers to censor transactions based on visible data or metadata associated to them.

The addition of new transaction types changes this visibility. In the case of a burn transaction, the value discarded is available to the network. In the case of a convert transaction, the value and asset type used in the conversion are available. In the case of a yield transaction, the value used to establish the later payout is available.

Each of these increases the risk of censorship. Network participants relaying pending transactions may use such information to decide if they wish to pass along a transaction to their peers, or simply discard it. Block producers may also use the information to decide if they wish to include a transaction in a block, which can deny a user access to converted funds or a later yield payout.

A related risk is that if a block producer chooses to selectively delay the inclusion of a transaction in a block, it can affect the value available to a user. Failing to process a burn transaction immediately keeps the supply higher, which may affect asset value depending on market behavior. Delaying the inclusion of a convert transaction means the block producer can examine the output of the oracle used to determine conversion rates, effectively exerting control over available value after the conversion takes place. Selective delay of a yield transaction could presumably affect the timing and value of the later payout, depending on the computation of the payout according to consensus rules.

These risks should be carefully considered and communicated, as they are challenging to mitigate whenever data or metadata are visible.

3.4.2 Protocol transaction structure

The information description of the protocol transaction design specifies that the transaction consists of a "single coinbase input" in addition to outputs constructed according to the protocol above. The role of this structure was initially unclear.

Salvium notes that this design is intended to better accommodate parsing tools, and does not otherwise impact consensus rules.

3.4.3 Use of return functionality

The informal description of the design of convert and yield transactions implies that the sender uses change outputs to compute the value $K_A^{o,r}$ used later by block producers, in the same manner described for return transactions. While this approach works and appears to meet its goals, it is inefficient. Specifically, it requires that the sender perform a subaddress table lookup using the method described in the return transaction protocol when performing the scanning process. This lookup will be for a change-based subaddress that is unique to each transaction, increasing the size of the lookup table. As discussed, this requires storage of more data than for a standard subaddress transaction. Further, it is not necessary for the sender to compute $R' = r'G$ by first sampling r' since it already has access to k_A^v to complete the required Diffie-Hellman shared secret.

Instead, the sender can simplify the construction of the additional data included in its convert or yield transaction. Specifically, it instead does the following:

1. Samples a group element R' uniformly at random.
2. Computes $K^{o,r} = \mathsf{H}_n(k_A^v R')G + K_A^s$.
3. Includes the values R' and $K^{o,r}$ in the transaction as additional data.

As before, the block producer uses R' as a transaction key and $K^{o,r}$ as a one-time output key in a protocol transaction. With this updated protocol, the sender may use its existing subaddress lookup table to recover the index associated to K_A^s during scanning.

3.4.4 Reuse of protocol transaction outputs

The Monero protocol, upon which Salvium is based, does not prohibit the reuse of one-time output keys between transactions.

This introduces risk if not handled safely. For example, if an attacker sees a pending transfer transaction, it can generate its own transaction reusing a one-time output key with a trivial value. If the user spends the attacker's output and later attempts to spend the honest output, the latter will be rejected by the network as a double-spend attempt, effectively burning the honest funds at little cost to the attacker. A common non-consensus mitigation is to ensure that wallets only show users the highest-valued output among any that share a one-time key, ensuring that an attacker could only supply more funds to the user than would otherwise exist. Other mitigations, like binding transaction context to one-time output keys, exist but are not in known use.

This risk still exists in Salvium, but in more complex ways due to the expansion of transaction types.

Because convert transactions result in protocol-enforced generation of outputs of different asset types in protocol transactions, it is not clear how the existing "highest-value" wallet rule would operate to decide which of a set of duplicated outputs to present to the user.

Additionally, because yield transactions result in later generation of outputs in protocol transactions that presumably increase supply in a controlled manner, duplication of one-time output keys could be used in an attempt to inhibit this inflation, albeit at a nontrivial cost to an attacker.

Generally, this overall risk exists because an attacker can produce a duplicate output key at one of three times: while the user's honest convert or yield transaction is pending, after the honest transaction but prior to the corresponding protocol transaction, or after the protocol transaction. The duplicate key may also be used in a different transaction type than the honest transaction.

Therefore, a combination of one or more consensus rules and non-consensus wallet mitigations may be helpful, depending on exact implementation:

- For any one-time output key specified by a convert or yield transaction, require an associated signature on the key as a consensus rule. Binding relevant transaction and output context to the signature could mitigate malleability or replay by an attacker, but at an added cost to transaction size and verification complexity.

- For any one-time output key specified by a convert or yield transaction (and possibly even burn or transfer transactions), have wallets bind context into the Diffie-Hellman exchange hashing operation used to produce the key. This could mitigate malleability or replay by an attacker, but may

still allow unwanted strict duplication that results in multiple associated protocol transaction outputs.

- For any one-time output specified by a convert or yield transaction, have wallets refuse to present receipt of funds directed to such outputs aside from those in corresponding protocol transactions. However, this may still allow an attacker to induce multiple protocol transaction outputs.
- Retain the functionality of the existing "highest-value" wallet rule for transfer transactions. If done carefully with respect to other transaction types, this may prevent an attacker from performing certain dusting attacks.