Haven Protocol code notes

Cypher Stack*

November 5, 2021

1 Introduction

This document describes technical information relevant to Haven Protocol. It reflects a snapshot of the codebase repository[1], but its findings and observations may be superseded by later changes. As with any code review, this analysis is limited and cannot capture all use cases or unintended behaviors of the codebase in all circumstances; it reflects a best-effort approach of the author to determine the extent to which relevant portions of the codebase reflect verification operations in the transaction protocol description[2] approved by its designers. It does not specifically address the scope of best-practice secure C++ design principles, for which the authors recommend further detailed review. The author asserts no warranty and disclaims liability for its use. The author further expresses no endorsement of Haven Protocol or its associated entities.

2 Structure

In this section, we describe the structural similarities and differences between the protocol description and its implementation.

2.1 Transaction types

The protocol description describes two types of transactions: *transfer* and *conversion*. Transfer transactions consume and generate assets of a single type, where balance is asserted if the combined value of spent inputs is equal to the combined value of generated outputs. Conversion transactions consume assets of a single type, and generate assets of the consumed type and a single additional type. In this case, balance is asserted if the combined value of spent input inputs is equal to the combined value of generated outputs after applying a scaling factor produced by a pricing oracle.

*https://cypherstack.com

[1] https://github.com/haven-protocol-org/haven-main/tree/v2.0.0-rc3

[2] https://github.com/cypherstack/haven-protocol-review/releases/tag/final

The protocol description requires that transfer transactions generate at least two outputs, which mitigates analysis based on knowledge of commitment masks or values. It requires that conversion transactions generate at least one output of each of the two required types, but discussions with the designers indicate the intent is to require at least two source- and destination-type generated outputs.

The implementation differs from the protocol in further delineating transaction types based on specific asset types. Transaction types are defined in `cryptonote::transaction_type`, and are inferred from given source and destination strings in `cryptonote::get_tx_type`:

- `UNSET`: no type has been specified
- `TRANSFER`: source and destination types are both `XHV`
- `OFFSHORE_TRANSFER`: source and destination types are both `XUSD`
- `XASSET_TRANSFER`: source and destination types are the same, but neither `XHV` nor `XUSD`
- `OFFSHORE`: source type is `XHV` and destination type is `XUSD`
- `ONSHORE`: source type is `XUSD` and destination type is `XHV`
- `XUSD_TO_XASSET`: source type is `XUSD` and destination type is neither `XHV` nor `XUSD`
- `XASSET_TO_XUSD`: source type is neither `XHV` nor `XUSD`, and destination type is `XUSD`

Any other combination of source and destination strings is rejected.

2.2 Asset types

Source and destination asset type strings for a transaction are established in `cryptonote::get_tx_asset_types`, where protocol consistency rules are applied based on the structure of consumed and generated transaction outputs:

1. The transaction must consume only a single asset type, which is defined to be the source asset type.
2. If the transaction is a miner transaction, the destination asset type is defined to be the base `HXV` asset type, and no source transaction type is specifically defined.
3. If the transaction is not a miner transaction and generates a single asset type, it must match the source asset type and is defined to be the destination asset type.
4. If the transaction is not a miner transaction and generates two asset types, exactly one of them must match the source asset type; both are defined to be the destination asset types.

5. If the transaction is not a miner transaction, it may not generate outputs of more than two asset types.

6. All source and destination asset types must belong to the static list defined in `offshore:ASSET_TYPES`.

These rules differ from the protocol description. Miner transactions, which are part of the parent transaction protocol, are not specifically defined in the scoped transaction protocol, but are restricted in implementation to the base asset; notably, miner transaction output types are not checked in this function to be of the base asset type; the destination asset type is set manually. Source and destination types default to an empty string, which is not specifically defined as either an asset type nor an indicated error condition.

2.3 Transfer transaction balance

The balance logic for transfer transactions of the types `OFFSHORE_TRANSFER`, `TRANSFER`, and `XASSET_TRANSFER` is established and checked in the function `rct::verRctSemanticsSimple2`. This function assumes prior establishment of source and destination asset type strings, which are checked against the static list of valid types. In the case of transfer transactions, the source and destination type are identical. The function is intended to process transfer and conversion transactions, and so we must ensure that the common logic applies to the specific protocol requirements of transfer transactions. Using the notation of the protocol description, the following is computed and checked:

$$\sum_{u=0}^{w-1} C'_u - \mathsf{Com}(f, 0) - \mathsf{Com}(f', 0) - \sum_{j=0}^{t-1} \overline{C}_j = 0$$

Here f is the transaction fee and f' is the *offshore fee* as noted in code comments. The offshore fee is not captured by the protocol description, but is handled in the same manner as the transaction fee. Transactions of type `TRANSFER` have the logic applied as the result of a default condition of a conditional block that checks against possible types; if another transaction type were presented that was not of the types in this block, it would evaluate using the transfer logic.

2.4 Conversion transaction balance

The balance logic for conversion transactions of the types `OFFSHORE`, `ONSHORE`, `XUSD_TO_XASSET`, and `XASSET_TO_XUSD` is also established and checked in the same function. In these cases, the source and destination asset type strings are different, but checked against the static list of valid types. Using the notation of the protocol description, the following is computed and checked:

$$\sum_{u=0}^{w-1} C'_u - \mathsf{Com}(f, 0) - \mathsf{Com}(f', 0) - \sum_{j=0}^{n_T-1} \overline{C}_j - \frac{1}{r} \sum_{n_T}^{t-1} \overline{C}_j = 0$$

Here, the value of r is scaled by an atomic unit factor `COIN` depending on whether `XUSD` is a source or destination asset type. As with transfer transactions, the offshore fee is not captured by the protocol description. Further, atomic unit conversions to and from `XUSD` type are not captured by the protocol description.

2.5 Burn amount verification

The conversion burn amount verification check required by the protocol also takes place in `rct::verRctSemanticsSimple2`. This is intended to confirm a plaintext amount sum for the source asset type corresponds validly to the associated commitments presented in the transaction. According to the protocol description, this is asserted by the following check:

$$\sum_{u=0}^{w-1} C'_u - \sum_{j=0}^{n_T-1} \overline{C}_j - \mathsf{Com}(f, 0) - \mathsf{Com}(0, a_T - \overline{a}_T) - \mathsf{Com}(y_T, 0) = 0$$

The check performed in the implementation differs in its commitment construction, performing this check:

$$\sum_{u=0}^{w-1} C'_u - \sum_{j=0}^{n_T-1} \overline{C}_j + \mathsf{Com}(0, \overline{a}_T) = \mathsf{Com}(y_T - f'', a_T) = 0$$

Here f'' is defined in the code as a scaled *burn fee* computed from the offshore fee, which is not captured by the protocol description. Note that while the commitments constructed in the implementation differ from the construction method in the protocol, they are algebraically equivalent (aside from the burn fee).

2.6 Mint amount verification

The conversion mint amount verification check required by the protocol takes place in the function `rct::checkBurntAndMinted`, which is called from the function `tx_memory_pool::add_tx2`. This is intended to check a claimed plaintext amount sum for the destination asset type. As noted in the protocol description, this is not checked against commitments presented in the transaction. It appears that a user generating such a transaction could simply provide a plaintext minted amount sum satisfying the check without any particular correspondence to commitments; however, we argue in the protocol description that security is not reduced by removing this check altogether. The verifier can produce the value on its own if needed, and avoid relying on the value provided in the transaction.

2.7 Linkable ring signature

Both transfer and conversion transactions require a linkable ring signature on each consumed source asset. The protocol description assumes the use of

CLSAG signatures for this use. The implementation uses the CLSAG implementation from its parent codebase, where verification of signatures is performed within `rct::verRctCLSAGSimple`, which binds an arbitrary message into the signature. The implementation of `rct::verRctCLSAGSimple` appears to be correctly ported from the parent codebase. While the protocol description does not specify the format of the message bound to each CLSAG signature, it should be checked that this message includes any replay-specific mitigations relevant to transactions that may differ from the parent codebase.

2.8 Range proofs

Generated outputs in both transfer and conversion transactions require one or more range proofs to assert committed values are within a protocol-specified range. The protocol description assumes the use of the Bulletproofs range proving system for this use. The implementation uses the Bulletproofs implementation from its parent codebase, where verification of aggregated proofs is performed within `rct::verBulletproof`. The implementation of the verification wrapper `rct::verBulletproof` (and its underlying direct verification function) appears to be correct as unchanged from its parent repository. Atomic unit conversion is not performed prior to range proof verification, as this is done for balance purposes separately.

3 Observations

In this section, we describe findings and observations within the Haven Protocol codebase relevant to verification in the transaction protocol, and note possible suggestions for improvement where appropriate.

3.1 Pricing oracles

The protocol relies heavily on a single pricing oracle that is assumed to securely provide responses to queries for asset conversion rates. No checks are assumed to be made about the consistency of its responses. The implementation relies on a set of hardcoded oracles `config::ORACLE_URLS` defined for the live network and test networks; for each network, the oracles share a single verification public key.

Oracle queries are performed in `Blockchain::get_pricing_record`. This function shuffles the oracle list (presumably for load balancing reasons), and then queries the shuffled list until a response is returned. The query includes a network version flag and encoded block timestamp. On receipt of a response, the function verifies a response signature against the hardcoded public key. If verification fails or no response is received from any oracle, an empty response is constructed that instantiates with zero values. The result is used to populate a pricing record on each supported asset type.

The oracle implementation currently deployed returns JSON data consisting of asset types with integer-valued conversion rates, a timestamp, an encoded signature, and unused auxiliary data. To verify the signature, a JSON string is reconstructed with all asset types, corresponding conversion rates, and timestamp (if nonzero). Signature values are decoded using a method that is not documented. The algorithm used to verify the signature is not made explicit, as it appears to be inferred from the encoded public key. Based on limited comments in the code, it may be ECDSA, but this is not confirmed.

All hardcoded mainnet oracles accept HTTPS queries on port 443, reject HTTP queries on port 80, and reject HTTP queries on port 443. Oracles are hardcoded by domain and not IP, appear to defer to the user's chosen DNS provider for IP resolution, and appear to defer to the user's local certificate authority for server authentication.

3.2 Pricing record inversion

In balance computation, pricing record values are converted to scalar field elements and may be multiplicatively inverted. The inversion algorithm is only guaranteed to succeed on nonzero input; there is a check within the `invert` function that will fail on zero input, but it only runs on the presence of a debugging flag. The pricing record values in balance computation are not tested within `verRctSemanticsSimple2` to assert they are nonzero. Pricing oracle queries may return zero values, or the entire query may fail and result in a zero-default pricing record instantiation. There exists a check in `tx_memory_pool::add_tx2` against zero-value specific entries within pricing records. A safer additional guard is to enable the inversion check on production builds; the operation is extremely efficient, and is the most direct guarantee that the algorithm succeeded. Another safe option is to test input scalars to the inversion function to assert they are nonero; this check would also be extremely efficient.

3.3 Common key type

The codebase, like that of its upstream parent, uses a single byte array type `rct::key` to represent two underlying mathematical types: elliptic curve group elements and elements of this group's scalar field. Both element types have canonical 32-byte representations, but they are not directly algebraically compatible. A comprehensive codebase update might use separate byte array types for curve group and scalar elements in order to reduce confusion and the likelihood of errors in algorithm design or implementation; however, this common type is widely used throughout the implementation, implying risk in such a wide-ranging change that would need to be carefully considered.

3.4 Inconsistent key initialization

Related to Observation 3.3, group and scalar elements are occasionally initialized inconsistently. The zero scalar is defined using the `rct::zero()` func-

tion, which returns a constant byte array `(0x00,...,0x00)` as its canonical byte representation. The zero (identity) group element is defined using the `rct::identity()` function, which returns as its canonical byte representation a constant byte array `(0x01,0x00,...,0x00)`. However, several locations in `rctSigs.cpp` initialize group elements unexpectedly using `rct::zero()` or `memwipe`, which could have unintended behavior if these values are not later set directly or by appropriate function calls to other algorithms or operations.

Examples of this behavior include:

- `rctSigs.cpp`, lines 933-935
- `rctSigs.cpp`, line 1328
- `rctSigs.cpp`, line 1584

Relying on later function calls or algorithms to reset group elements to another value is not robust, and may lead to unsafe results. Care should be taken to initialize group and scalar elements correctly when using `rct::key`, as it is not always obvious from context which element type is intended.

3.5 Inconsistent asserts

Low-level balance verification in `verRctSemanticsSimple2` performs computations on input and output data based on transaction and asset types discussed previously. For this reason, it is essential that these types are valid and consistent. The asserts contained in the low-level balance verification check that source and destination asset types are valid by their inclusion in `offshore::ASSET_TYPES`. Later, the balance computation sorts outputs by asset type, and then uses the transaction type to complete the computation. However, this does not account for the required consistency between output asset types, source and destination asset type strings, and transaction type provided as function arguments. Rather, these consistency checks are offloaded. For this reason, the inclusion of one consistency check for valid source and destination asset type strings appears inconsistent, as it is also performed prior to the function being called. It may be useful to standardize the validity and consistency checks between transaction types and asset types to improve reliability and efficiency, and reduce the possibility of error. In particular, if these validity checks are not intended to be used in other parts of the protocol, it may be safest to call them directly from `verRctSemanticsSimple2` (to ensure they are run whenever this verification takes place), or to document carefully the requirements and assumptions on the function arguments.

3.6 Underspecified transaction type check

When performing the balance check in `verRctSemanticsSimple2`, the transaction type is crucially used to identify the scaling and outputs to be used in the calculation. This is done by building the sum-to-zero element `Zi` in a way that

is dependent on the transaction type. Generic `TRANSFER` transactions are not specifically checked by the conditional in balance verification, instead being the result of a default condition.

While this does not pose a problem if possible valid transaction types are not added, it would be safest to reject transactions not meeting the expected type, as there may be unintended behavior resulting from attempting to complete the computation and balance validity check on an unexpected transaction type.

3.7 Unnecessary identity computations

As noted in Observation 3.4, the elliptic curve group identity element can be obtained via a call to `rct::identity()`, which returns a constant byte array consisting of the canonical representation of this element.

In several locations in `rctSigs.cpp`, the identity group element is computed by computing the scalar-group multiplication `scalarmultH(d2h(0))`, which is unnecessary and less efficient.

Examples of this behavior include:

- `rctSigs.cpp`, line 1327
- `rctSigs.cpp`, line 1583
- `rctSigs.cpp`, line 1866
- `rctSigs.cpp`, line 1869
- `rctSigs.cpp`, line 1870
- `rctSigs.cpp`, line 1871

All such function calls can safely be replaced with `rct::identity()` calls for clarity and speed.

3.8 Fee ranges

The protocol description requires that fees be checked to lie within a protocol-specified range. Because fees are presented in the clear, they do not come equipped with range proofs. However, fees are used in the verification function `verRctSemanticsSimple2` to produce commitments that may overflow the balance check if not asserted to be within the required range. All fees, including those outside of the scope of the protocol description, should be carefully checked to ensure they cannot overflow the balance check. One way to ensure this is by suitable truncation encoding that is properly re-encoded to a scalar value for commitment computation.

3.9 Undocumented signature verification

As noted previously, pricing oracle responses come equipped with a signature that is verified prior to acceptance using `pricing_record::verifySignature`. However, the details of this verification process are not clearly documented.

In particular, the signature algorithm intended to be used for the public key appears to be inferred from the key type, which is not specified in the code. Second, a comment indicates that the decoded signature is reformatted to match an OpenSSL format, but the details of the formatting operation are unclear, so it is not possible to properly assess. Finally, the verification algorithm is unspecified at the time of verification (related, presumably, to the key type as noted above).

Because price oracle response validity is crucial to the safe operation of the protocol, the methods used to verify oracle signatures should be clearly documented.

A separate option (if reliance on OpenSSL functionality is not desired) may be to use a Schnorr signature on the oracle response, which could take advantage of structures and functions already present in the codebase, including those for deseralizing keys and signatures. However, this change could introduce additional implementation risk and complexity.