CashFusion

Authors: Jonald Fyookball, Mark B. Lundeberg

Introduction

THE PROBLEM:

CashShuffle is a powerful tool for obfuscating the origin of a coin. However, after shuffling a wallet, a user will inevitably wish to consolidate several coins, and for this another tool is needed. We need a method to coordinate coinjoin transactions with multiple inputs per user. This is inherently challenging because we want to hide input linkages while simultaneously attempting to blame/ban users who don't sign all their inputs.

THE SOLUTION:

We propose a blind verification scheme in which each input and output is validated by a random player, using a series of cryptographic commitments that allow an uncooperative participant to be identified and banned.

CashFusion provides high levels of privacy via a flexible scheme that allows an arbitrary number of inputs and outputs of non-standard amounts. It provides anonymous, trustless coordination with usually zero-knowledge of linkages revealed to other players or the server.

PART I: Protocol Overview

To review the big picture, the final result is a large coinjoin transaction with many participants and many inputs and outputs. In the simplest form, this is achieved by players covertly sending their transaction components (inputs and outputs) to the server. The server then shares all the information with all players, the players sign all their inputs, then (covertly) send them back to the server, and finally the server again shares with everyone so that all players have everything necessary to assemble a valid transaction.

But, in order add the blame capabilities (to allow banning a user who didn't sign all her inputs), each player first creates a salted hash of each of his inputs and outputs, and sends them to the server. Once all the players have joined the pool and have sent their commitments, they then proceed to covertly announce their transaction components to the server over TOR.

If there is a problem with the transaction, each component is assigned to another player at random for verification. The owner of the component sends the secret information, encrypted with one of the communication keys that's paired with each commitment.

To prevent players from cheating by using the same component in multiple commitments (by hashing it with a different salt), the salt value itself is also hashed and paired with the covert announcement of the transaction component.

If there is a problem during verification, the verifier reveals his private communication key to the server, and the server determines whether to blame the accused or the accuser.

Finally, to prevent DOS attacks, the blame mechanism will kick out the bad player and attempt the round again

with one less player. The process can be repeated as many times as desired.

Inputs

The scheme described so far would generally be sufficient to provide trustless coinjoins with multiple inputs. However, it would have strict limitations of requiring fixed input amounts and a single output.

The challenge with variable input amounts is ensuring every player contributed sufficient funds to cover their outputs.

To solve this, a set of Pedersen commitments are utilized. Using its homomorphic property, the server can ensure the sum of inputs and outputs for each player is zero, without knowing any of the values. And, each input or output that is verified will be also cross-checked against its paired amount commitment, thus linking the component commitment to the amount commitment in a simple way.

Inputs are represented by a positive integer indicating the number of satoshis, and outputs by a negative integer. Fees are handled elegantly by subtracting a standard amount from an input or adding a standard amount to an output. For example, if an input is 10,000 satoshis and the standard fee for an input is 150 satoshis, the Pedersen commitment should be for 9850. This idea can be extended to allow players to add extra fees by relaxing the rule to allow the commit to be less than or equal to the expected value. (e.g. x <= 9850).

Outputs

Allowing a flexible number of outputs presents its own challenge. Whereas inputs could be signed over all outputs and "extra" inputs ignored, extra outputs would invalidate the transaction and are hard to detect and blame when announced covertly.

To solve this, we utilize 2 more puzzle pieces: blind signatures, and "blank" placeholder components.

Here, blind signatures function like a submission "token" that is required with ALL covert announcements (not just outputs) , as follows: Players blind a message (the message is the transaction component plus the salt commitment) and present the blinded message to the server for signing. Since it is blinded, the server doesn't know the actual contents of the transaction component.

When attempting a covert announcement, the player unblinds the message and the server can verify if their signature matches the message. The server is prevented from learning any linkages but it does set a known limit on how many "components" each player is submitting.

Next, we set a requirement that each player must covertly announce exactly 23 components. Why 23? It could be anything, but this allows for example 8 outputs and 15 inputs. By forcing each player to commit to exactly 23 components, this prevents Alice from submitting anything extra because the server won't accept the same signature more than once and will disallow the covert announcement.

Since a player normally will use less than 23 inputs and outputs, the remaining components are considered "blanks" but still need some tangible form: they still need to be a data string that produces a salted hash, etc. For example, "BLANK33ed2a3a280a549a672b". These can be randomly generated by the user and are

assigned an amount value of zero. They are verified like any other transaction component.

Therefore, Alice is prevented from covertly announcing a transaction input or output and not including it in her component commitements because then she'll be missing a component somewhere else.

PART II: Protocol Phases

Note -- this section doesn't describe the exact serialization or parsing rules. Generally, `Hash` means a SHA-256 hash. Pedersen commitments are elliptic curve points on secp256k1, sent uncompressed for easy summation. Other pubkeys (also on secp256k1) will typically be sent compressed.

Phase 1. Setup and Waiting List

Clients connect to the server and download a list of parameters:

- The a list of possible output tiers that the client may register for. Example: [10000, 20000, 50000, 100000, 200000, ...]
- The required fee rate for components. Example: 1.000 satoshi/byte
- The minimum and maximum excess fee paid per player (see phase 3 below for definition of excess fee). Example: 11 satoshi minimum, 300000 satoshi maximum.
- Other possibly important parameters.

The clients then attempt to construct a fusion contribution for each tier: for their given inputs, make a list of random outputs that sum to those inputs less fees. If the tier size is too large or too small, this process will fail (too many or too few outputs). The client then indicates to the server, which tiers are of interest.

Each tier forms a separate 'waiting pool', though a client can wait in multiple pools simultaneously. Once enough players have joined a given pool, the server moves pulls those clients out of all pools, and then moves the clients into the fusion process at the filled tier.

Phase 2. Start Round

(**MESSAGE 2** ~ 850 bytes) In this phase, the server replies back to the player who just joined and shares:

1. The *round public key* that the server will use for the blind signatures.
2. 23 unique nonce points per player which will be used for blind signatures on the components.
3. The location where covert submissions should be later made (host / port).

The exact timing of message 2 is critical, as it sets the timeline for the rest of the protocol. Immediately once clients have received this message, they should start trying to establish covert connections to the submission point via Tor, in background. We define the time of client's receipt of this message as *TC*. Below, various time limits are described as, e.g., "*TC* + 5" meaning "5 seconds after client received message 2".

Since clients will expect to receive messages by various times, the server needs to exclude any clients that move too slowly. In order that the various clients' *TC* are not too dispersed in time, the server should attempt to

simultaneously send message 2 to each client at time *TS*, and kick any client that does not respond with a message 3 (see below) within 2 seconds. We will refer to time *TS* as well below, when the server needs to kick slow clients.

Phase 3. Player Commitments

By this point, the client has already determined the set of 23 components which they will submit later on. A component may be one of the following three types:

- A transaction input, contributing satoshis to the transaction: `component = ( Hash(Salt), prevout_txid, prevout_num, pubkey, amount )`.
- A transaction output, spending satoshis from the transaction: `component = ( Hash(Salt), output script, amount )`.
- Blank: `component = ( Hash(Salt) )` -- this doesn't contribute to transaction, but helps to obfuscate the number of transaction components submitted by each player.

The random Salt value makes each component unique and also is used in the component hash commitment, below.

(**MESSAGE 3** ~ 4 kB) The client sends the following to the server:

- For each of the 23 components, the client shares a pair of commitments, as well as a communication key that will come into play during the blame phase:
    - A *component hash commitment* to the serialized content of the component: `Hash( Salt, component )`.
    - A *component pedersen commitment* to the net satoshi amount added/subtracted by the component:
    - For an input, this is the UTXO's value minus the required fees `pedersen_amount = input_amount - fees` where `fees = feerate * input_size` rounded up. For a P2PKH with compressed public key, `input_size` is 141 bytes when signed with Schnorr.
    - For an output, this is negative of the UTXO's value, also minus the required fees: `pedersen_amount = - output_amount - fees`, where `fees = feerate * output_size` rounded up. For a P2PKH output, `output_size` is 34 bytes.
    - For a blank, `pedersen_amount = 0`.
    - A *communication public key*, to which only the client knows the private key.
- The client reveals the *summed* nonce `k_total` and the *summed* amount `pedersen_amount_total` to the server. Note that this `pedersen_amount_total` will be equal to the client's *excess fee* -- the total fee contributed, minus the per-component fees mentioned above.
- The client chooses a random 256-bit number `X` and sends its hash: `Hash(X)`.
- The client prepares 23 blind signature requests from the the round public key, on each component.

The server validates the following. Violating clients are kicked immediately.

- A particular component hash commitment must not be duplicated, either in a single client's submission,

or between different clients' submissions.
- The server sums up the component pedersen commitment points. The sum must match the revealed sum-nonce and sum-amount.
- The excess fee (i.e., `pedersen_amount_total`) needs to satisfy the minimum and maximum limits. There are a few reasons for these limits:
    - Besides component input/output sizes, transactions have a small overhead (53-57 bytes: version, locktime, and input/output counters). If the component feerate is exactly 1 sat/byte, then the transaction would fall just short of the 1 sat/byte relay rate. Hence a small excess fee minimum is needed per player (just 11 sats per player is sufficient, if a fusion round never runs with less than 5 players).
    - In order to discourage non-participating clients that just contribute blanks, it is helpful to require a nonzero excess fee per player. Any tiny nominal value (even just 1 sat) is sufficient to cause those clients to be detected and blamed after the first round.
    - Some systems refuse to broadcast transactions that exceed a certain 'absurd fee' level. In ElectrumX attached to Bitcoin ABC, this limit is 0.1 BCH. Thus it is a good idea to impose a maximum per-player excess fee, that makes sure the overall transaction fee is not too high.
- Message 3 should be received by *TS* + 3.

Phase 4. Sharing Blinded Signatures

The pool should be ended if too few connected clients remain at this point.

(**MESSAGE 4**) The server Upon receipt of message 3 above from all players, the server completes the 23 blind signature requests and returns them to the clients. The clients can then unblind the signatures.

Clients expect to receive message 4 by *TC* + 5.

Phase 5. Covert Announcements of Inputs and Outputs

In this phase, each player covertly sends 23 separate messages over 23 separate covert communication channels (which started to connect in phase 2 above).

(**MESSAGE 5** ~ 100 to 200 bytes) The covert message:

- Explicitly or implicitly identifies which fusion pool it is intended for.
- Contains a `component` as defined in phase 3 -- note that this includes the transaction data as well as `Hash(Salt)`.
- Contains the unblinded signature over `component`, which verifies against the round public key.

(**MESSAGE 5b** ~ 10 kB to 100 kB) Simultaneously, the server shuffles the full list of (*component hash commitment*, *component pedersen commitment*, *communication public key*) tuples from all players, and shares it to all players. This one of the larger messages in Cash Fusion and so it helps to give clients time to download.

These covert submissions happen between *TC* + 5 and *TC* + 10. The server stops accepting covert submissions at *TS* + 15, and times out any client that hasn't finished receiving message 5b by *TS* + 15.

The server checks each submission as it arrives. * There must be a valid signature ove`component`, from the round public key. * The component's `Hash(salt)` must be length 32. * Inputs must have a valid txid (length 32) and have a valid pubkey (uncompressed or compressed). * Outputs must have standard scripts (P2PKH or P2SH) and the amount must satisfy the dust limit.

Invalid submissions are discarded. For valid submissions, the component is recorded. Note that if duplicate components are received (same salthash and same component data), the duplicate is silently discarded without error (this could be a redundant submission due to a communications failure).

Phase 6. Sharing Components

Once the server has received all the transaction components, the pool should be ended immediately if the resulting transaction would be unsuitable for privacy (too few inputs / outputs, too few reasonable-looking amounts are present, ...). The pool should also be ended if too few connected clients remain at this point.

The server performs further sanity checks on the component list, to determine whether the signing phase should be skipped outright:

- Check the number of valid submissions, which should be exactly 23 * num_players.
- Check the transaction fee -- the total fee should be equal to the sum of input/output per-component fees (per the formulas in phase 3), plus all the excess fees declared by clients. If the transaction fee does not match, then it means one or more clients is lying in their pedersen commitments (or have failed to submit components), and the transaction signing phase will be skipped.
- [Optional] The server can consult the blockchain and make sure the inputs exist properly (and are confirmed).

Note that there are certain detectable component errors (such as duplicated input (txid, num) pairs) that require moving to the signing phase.

(**MESSAGE 6 ~ 10 kB to 100 kB**) The server performs the above checks and sends this message at*TS* + 15:

- Full list of received components, either in sorted or shuffled order.
- A boolean indicating whether to skip signing the transaction (message 7A).

Clients expect to see message 6 by*TC* + 20. Again, this is a large message so some download time is allowed for. They must then check that the component list includes all their submitted components, and disconnect if this is not the case.

Phase 7. Covert Announcements of Signatures

At this point, the players and server can compute the*session hash*. This is a hash of all shared knowledge, and helps prevent shenanigans if the server were to tell different information to each player. The session hash commits to:

- the string b'Cash Fusion Session',
- the tier size,
- the round pubkey,

- the covert submission host and port,
- the full list of commitments (message 5b)
- the full list of components (message 6)

The transaction is then assembled by starting with an empty input list and an output list initialized with a 0 sat output paying to a 34-byte scriptpubkey: `0x6a 0x20 <32 byte session hash>` (i.e., `OP_RETURN PUSH(session hash)`). Then, iterating in order through the component list: append inputs and outputs as they come, and skip blanks. The final result is not sorted.

Both the client and the server assemble the transaction in this way: the client to produce transaction signatures, and the server to verify them.

(**MESSAGE 7A** -- covertly) Clients then prepare Schnorr transaction signatures (with sighash byte 0x41, i.e., SIGHASH_ALL | SIGHASH_FORKID) for their inputs, and submit them covertly (over the same connection as used for the input in phase 5, preferably). The covert message:

- If necessary re-identifies which fusion pool it is intended for.
- If necessary re-identifies which component it is for.
- Contains the 65-byte Schnorr transaction signature.

These covert submissions happen between *TC* + 20 and *TC* + 25. The server stops accepting covert signature submissions at *TS* + 30. As each submission comes in, the server verifies the transaction signature before storing it.

Phase 8. Executing the Transaction

At *TS* + 30, the server knows whether the transaction is complete (all signatures received) or not.

If the signing step was not skipped: * The server should mark all inputs with missing signatures as bad. * The server should examine the inputs to see if there are any duplicates (same outpoint being spent *with same claimed pubkey*). If so, then only mark an input as bad if there is another duplicate input that has a valid signature. (i.e., if one is signed and one unsigned, then only one is bad. If both are validly signed or both are unsigned, then mark both as bad.) * It's important to require pubkey matching, since (until this point) someone can claim a utxo they don't own and provide a valid signature, just with the wrong public key. Such liars will be eventually caught during blame phase but that requires looking at the blockchain.

If signatures are complete and no components are marked as 'bad', then the server attempts to broadcast the transaction. If this succeeds, then the fusion is complete and no more work is necessary -- the server sends 'success' to clients then disconnects them. In all other cases the server sends a 'fail' message.

[Optional] The server can scan through the inputs to see which one is inconsistent with the daemon, and mark that component as bad.

(**MESSAGE 8**) Result of fusion:

- A boolean indicating "success" or "fail".
- If success, a full list of the inputs' signatures.
- A list of all the bad component indices.

Notably, broadcast can unfortunately fail if an input is inconsistent with the state of the bitcoin daemon (blockchain or mempool). This can happen for a variety of reasons: The input's amount was wrong, input's scriptpubkey is not p2pkh with matching pubkeyhash, or the input coin is missing / already spent / an unmatured coinbase.

Clients expect to receive a 'success' or 'fail' message by *TC* + 35. If any of the client's own components are marked as 'bad', they should disconnect immediately instead of proceeding with blame. Clients should also disconnect if the bad components list looks weird: decent blank/output components marked bad, or the number of remaining good inputs would be too small to result in a good fusion, in the next round.

Phase 9. Sending and Sharing Proofs

Now, the blame phase begins. Clients are now required to send proof of the correctness of their actions to another client, that they know via the communication key observed in Message 5b.

The destinations of the proofs are sent randomly, seeded by the number *X* that the client provides. The algorithm is:

- Start from the full list of commitments (in message 5b), and remove the client's own commitments from this list.
- Let N be the length of this reduced list.
- For i = 0 ...:
  - calculate `sha256(X | <i encoded as a 32-bit integer> )`
  - truncate to the first 8 bytes, and convert to a big-endian 64-bit integer X.
  - calculate `pos = (X * N) >> 64`. This will be nearly uniformly distributed between 0 ... N - 1.
  - the client is then responsible for sending a proof for their commitment `i` to the commitment with position `pos` in the reduced list.

Because the process is deterministic, the client and the server both know which of the client's commitments will be going to which destination. However, the server will inform each verifier about what messages to expect.

(**MESSAGE 9A**) The client sends a series of encrypted proofs, encrypted to the communication keys of the recipients.

- The client reveals the random number *X* commited to in Phase 3.
- The client provides 23 encrypted proofs, each regarding its respective commitment from phase 3. Each proof plaintext contains:
  - The index of the component in the all-components list (from message 6).
  - The `Salt` value that was used in both the `component` and the component hash commitment (from phase 3).
  - The revealed pederesen nonce and amount for the component pedersen commitment (from phase 3).

The encryption algorithm is as follows: * Create an ephemeral nonce and corresponding secp256k1 poin `Y`

and perform a diffie-hellman shared secret derivation.. * The 256-bit symmetric encryption key is then given by: `K = SHA256(compressed diffie hellman point)`. * The proof plaintext, `[4 byte length][serialized proof ~ 73 bytes long][NUL-padding up to 80 bytes length]`, is encrypted using AES256 in cipher block chaining mode, yielding the 80 byte `ciphertext`. * A message HMAC is calculated as `H = first 16 bytes of HMAC_SHA256(K, ciphertext)`. * The full encrypted proof is then `(compressed Y, ciphertext, H)`. It is decrypted by first deriving `K`, then verifying `H` (using a constant-time compare), and then finally decrypting the ciphertext.

In case a communication key is invalid (and a player cannot perform the diffie-hellman shared secret), the player may simply submit a blank entry for that encrypted proof. The recipient cannot blame this (since they cannot provide the private key), and the server permits empty proofs.

The server expects to receive all proofs by *TS* + XXX (kicking the player if any are missing). After checking that *X* is correct, the server then relays the proofs:

(**MESSAGE 9B**) The server relays each encrypted proof verbatim to the intended destination client (who owns the communication key), along with information about its source:

- The list of relayed proofs
    - The encrypted proof, verbatim.
    - The index of the to-be-proven commitment in the all-commitments list (from message 5b).
    - The index of the recipient's communication key that will decrypt the proof (from phase 3).

Note that multiple proofs can be sent to the same communication key, and this is common: for a given key, there is a 37% chance of no proofs, 37% chance of one proof, 18% chance of two proofs, 6% chance of three proofs, etc..

Phase 10. Blame

Clients should then examine the received proofs, then send a 'blame' proof to the server if any problems are identified. The following reasons can induce a blame:

- The proof is corrupted (cannot decrypt / parse).
- The server marked the component as 'bad', or the client identifies something obviously bad about the component (though the server should have picked this up already).
- The salt doesn't match the `Hash(Salt)`.
- The salt doesn't produce a matching component hash commitement.
- The provided pedersen nonce doesn't yield a matching commitment.

(**MESSAGE 10A**) * The client indicates which of the proofs from message 9B is problematic. * If the encrypted proof can be decrypted (the HMAC is correct) but there are any other issues, the client sends the symmetric key. This only reveals the one message. * If the encrypted proof cannot be decrypted at all (the HMAC does not match), the client sends the private communication key. Probably this will inadvertently reveal other innocent proofs to the server as well (42% chance!). * The client can include a textual message explaining why the blame is deserved. This is useful for debugging in case the server doesn't agree about the blame. * A boolean indicates that the commitment and component data were perfect, but the client only found problems after consulting the blockchain.

Upon receipt of a blame, the server first checks that the blame is valid, then examines the proof. * If it includes the private key, then the private key must map to the communication pubkey originally provided, and the decryption of the proof using this private key must *fail*. * If it includes the session key, then the proof must decrypt successfully. The server then reproduces the entire examination process and finds any issues. In case of issues, the offender is kicked.

Phase 11. Restart

By this point, the server has hopefully found at least one client to blame. It is possible however that nobody was found to blame:

- If the owners of verifying communication keys have disappeared for some reason before message 10a, then unfortunately nobody is around to decrypt some of the proofs, and they can only be given the benefit of doubt.
- Malicious or lazy clients may opt to issue no blames even for bad proofs.
- Inconsistencies with the blockchain, especially involving missing / spent transaction inputs, are unfortunately a bit more subjective in nature. If inputs are unconfirmed then differences in relay policy might make the transaction relay on one node but not another. The server and client might be on different chain tips (perhaps a block is being propagated just at this time) and see different confirmed coin sets.

PART III: Discussion of Protocol Design

The ideal goal of CashFusion is to allow trustless coinjoins with arbitrary amounts and values for inputs and outputs (obviously respecting the rule that the sum of a player's outputs cannot be less than the sum of his inputs), with each player revealing zero knowledge to any other player, or to the server. (The impetus to prove one's correct behavior here is implied, as it is the key to blame and anti-DOS.)

Zero Knowledge

Theoretically, a pure ZKP approach is impossible. Since by definition nothing is revealed about any player's transaction components, there is nothing preventing Alice from colluding with Bob and sharing an input. In other words, there is no information about any common elements in the sets of components of 2 players. Thus, exclusion or uniqueness can't be proven in zero knowledge.

The perfect solution would therefore be some form of multi-party computation (mpc) in which the goal is fulifIled whilst making it computationally infeasible to gain any knowledge. However, this is a difficult problem and thus as a practical matter, we have instead chosen to create an approximate solution using a clever combination of some more basic tools, including garden variety hash functions.

The Basic Approach

The approach in CashFusion is to have each player make a cryptographic commitment to each of their transaction components (both inputs and outputs). When there is a problem with the transaction, each player proves each component individually to one of the other players in the group, chosen randomly.

Each player's list of commitments is a grouping that identifies a player that can be banned, but another key ingredient in CashFusion is that this list is only known to the server. This prevents the players from learning anything about linkages, no matter how many commitments any of them verify.

The server doesn't participate in the verification process itself, so it also learns nothing. Some collusion is possible between a malicious server and players it controls, but even in that case, the information leakage is minimal. We will elaborate on this momentarily.

Design Trade-Offs

When making design choices in this situation, there is a trade-off between patching tiny security holes vs adding complexity and unreliability in the form of more protocol phases. One must first understand that even with a perfect fusion protocol, sybill attacks are possible and can be taken to the extreme if the server is malicious. (Many of the trade-offs center around the server). When judging the merits of a security trade-off, it is helpful to compare what the security hole allows versus the existing attacking vectors, which may be already larger attack surfaces.

Returning to the discussion about collusion between a malicious server and a player it controls: In CashFusion, a player who is randomly verifying other players' transaction components could report all that information to the server, and the server could cross-reference this with its commitment lists from the players, looking for instances of multiple components on the same player's list.

As a result, the server may learn a few random linkages, and this is more than what two colluding players would learn in a perfect protocol. (Multiple rounds of shuffling and fusion could likely render this kind of attack inert.) As more colluding players are added in the current protocol, more linkages would be learned, contrasted with more colluding players being added in a perfect protocol, which only reveals more probabalistic taint information.

Therefore, this imperfection in CashFusion is manifested as only a moderate increase of information leakage to an adverserial group. Moreover, the additional leakage requires a malicious server, which already has the more powerful weapon of extreme sybilling. This means it is not worth attempting to patch this hole, particularly when doing so would mean a fundamentally more complex setup that doesn't rely on the server as much. (See Appendix B).

MITM Attacks

Another class of security considerations consists of various Man-in-the-Middle (MITM) attacks from the server, designed to reveal more information about the players' transaction components to the server.

One solution that was implemented is the use of an OP_RETURN output to store some global-state information. Although using OP_RETURN in this way may be considered inelegant, it broadly handles many problems. For example, if the server were to maliciously give each player a different public key for the blind signatures, it would be able to group inputs together. Unfortunately, including the key inside a component commitment doesn't work because we might not reach the blame phase, making the bad behavior undetectable. An alternative solution to OP_RETURN would be to check the commitments anyway either before or after broadcasting the transaction, but this would make the protocol more complicated and slower.

In addition to the blind signatures, the communication keys (used for encrypting a verification message) could also be spoofed. Here, the OP_RETURN prevents the server from executing the MITM attack in a non-disruptive manner, which would be most dangerous.

But if the protocol reaches the blame phase, the server could use previously committed fake transaction components, causing each user to prove all their information to the server. This would require the server to create a number of transaction inputs, although would take less effort than an extreme sybill attack.

In the aim of reducing complexity, we allow this attack vector as a limitation for now. It seems the best way to patch this in a future version will be to add client-side logging to detect high failure rates for servers. In addition, we can take advantage of the birthday paradox (there is a high probability that any players I'm verifying are also verifying me.) This makes the attack almost impossible without directly sybilling or making the attack detectable by the absense of birthday collisions between verifiers.

It is also possible for the server to spoof commitments, causing a player to blame a non-existent player, which would reveal the private key of the commitment. But this is of minor concern since there is only 1 blame per player per round anyway. The most the server could do would be to learn 1 component from each player per round and attempt to build information based on exclusion over multiple rounds.

If this ever becomes a real concern, it could be handled by adding additional layered encryption between players before blame is issued; the extra complexity is not worth it.

As a final point of discussion regarding MITM attacks, the public keys of the transaction inputs themselves may initially appear to be the best method of authentication, but that doesn't seem to work as it's part of the same information we need to keep hidden.

Avoiding Amount Linkages Through Combinatorics

Naive multiparty coinjoin schemes are vulnerable to chain analysis based on amount linkages. For example, if you see a joined transaction of (0.5, 0.5, 0.4, 0.7) -> (0.4, 0.6, 0.3, 0.8), this can be uniquely decomposed into (0.5, 0.5) -> (0.4, 0.6) and (0.4, 0.7) -> (0.3, 0.8). (references: https://www.coinjoinsudoku.com/, https://github.com/Samourai-Wallet/boltzmann).

As a result of the above analysis, modern coin shuffling schemes have focused on making equal-amount coins, which intrinsically are indistinguishable (CashShuffle, Wasabi, etc). In isolation, these shuffle schemes are essentially perfect, especially since the cryptographic protocol allows parties to hide information even from each other.

Unfortunately, the production of equal-amount coins is impractical for various reasons. Foremost, it has a "toxic waste" problem: producing identical amounts leaves unmixed and fully-linkable change in the majority of cases. Although the change coin can be successively shaved down in size, this requires many steps and also leaves behind a string of weakly linked shuffled coins. If those shuffled coins are all spent together, the weak linkage becomes a strong linkage (reference: https://cointelegraph.com/news/samourai-wallet-wasabis-coinjoin-management-lacks-privacy).

Moreover, equal-amount outputs provides no easy avenue to private consolidation. A joined consolidation of (0.5, 0.5, 0.4, 0.6) -> (1.0, 1.0), while now providing indistinguishable amounts, has now unfortunately linked the

inputs together: (0.5, 0.5) belongs to one party, and (0.4, 0.6) to the other. And, due to the repeated shaving process producing many unspent coins, consolidation is highly desirable. Even simple re-shuffling to increase the small initial anonymity set is difficult since real transactions pay fees; you can't shuffle a 1.0 coin to another 1.0 coin, rather you need a bit more on the input.

In CashFusion, we have opted to abandon the equal-amount concept altogether. While this is at first glance no different than the old naive schemes, mathematical analysis shows it in fact becomes highly private by simply increasing the numbers of inputs and outputs. For example, with hundreds of inputs and outputs, it is not just computationally impractical to iterate through all partitions, but even with infinite computing power, one would find a large number of valid partitions.

Consider a transaction where 10 people have each brought 10 inputs of arbitary amounts in the neighborhood of ~0.1 BCH. One input might be 0.03771049 BCH; the next might be 0.24881232 BCH, etc. All parties have chosen to consolidate their coins, so the transaction has 10 outputs of around 1 BCH. So the transaction has 100 inputs, and 10 outputs. The first output might be 0.91128495, the next could be 1.79783710, etc.

Now, there are 100!/(10!)^10 ~= 10^92 ways to partition the inputs into a list of 10 sets of 10 inputs, but only a tiny fraction of these partitions will produce the precise output list. So, how many ways produce this exact output list? We can estimate with some napkin math. First, recognize that for each partitioning, each output will typically land in a range of ~10^8 discrete possibilities (around 1 BCH wide, with a 0.00000001 BCH resolution). The first 9 outputs all have this range of possibilities, and the last will be constrained by the others. So, the 10^92 possibilies will land somewhere within a 9-dimensional grid that cointains (10^8)^9=10^72 possible distinct sites, one site which is our actual output list. Since we are stuffing 10^92 possibilties into a grid that contains only 10^72 sites, then this means on average, each site will have 10^20 possibilities.

Based on the example above, we can see that not only are there a huge number of partitions, but that even with a fast algorithm that could find matching partitions, it would produce around 10^20 possible valid configurations. With 10^20 possibilities, there is essentially no linkage. The Cash Fusion scheme actually extends this obfuscation even further. Not only can players bring many inputs, they can also have multiple outputs.

In the future, if desired, the combinatoric effect can be improved; it's easy to 'fuzz' the link between input and output totals by having fees be randomized or having outputs be quantized to more than 1 satoshi resolution.

Expected usage

Wallets can repeatedly use CashFusion to mix multiple inputs to multiple outputs, to increase the anonymity set. We expect this regular 'churn' of re-mixing to provide a strong foundation of anonymity set and liquidity.

Initially, wallets contain few utxos and so will primarily use Cash Fusion to break these utxos apart. Once a wallet contains more utxos that are delinked, it can start to make Cash Fusions where a few inputs are brought in, but still outputting several coins. At equilibrium, a wallet would contain around 100 coins, and randomly choose ~10 inputs to mix together into ~10 outputs.

Appendix A. Proof of Soundness

Proof of Soundness

We wish to prove that **zero knowledge (beyond what is publicly available) is revealed about any of the players' inputs or outputs** during the fusion process. It works because the linkages between players' commitments are only kept by the server, and the server doesn't participate in verification.

Let *X* be the set of a player's transaction components and commitments in this scheme. Let *a* represent a transaction component and *b* its commitment. We identify the basic unit of knowledge as showing that 2 components (inputs or outputs) belong to the same player.

$a1 \in X, a2 \in X$

A component and its commitment form an ordered pair (a,b).

If two commitments are known to belong to the same player and it also known which components the commitments correspond to, then we can trivially deduce that the two components belong to the same player.

Given (a1,b1), (a2,b2): $(b1 \in X, b2 \in X) \rightarrow (a1 \in X, a2 \in X)$

For the Players:

Since the distinct list of each player's component committments are kept on the server and not shared with the players, the players have no knowledge of the linkages and cannot state $(b1 \in X, b2 \in X)$.

Let *Y* be the total set of all the players' components. Since no subsets (ownership of components) is announced, no knowledge of linkages exist.

$(X \subset Y, a \in Y) \rightarrow (a \in X = ???)$

For the Server:

The server knows the linkages of the commitments but does not participate as a verifier and thus doesn't learn the link between the component and its commitment. The exception is the counterblame phase where one component and its commitment are revealed, but one pairing is insufficient to gain knowledge.

Given only (a1,b1): $(b1 \in X, b2 \in X) \rightarrow (a1 \in X)$, but $(a2 \in X = ???)$

Appendix B. Sketch of an MPC-based Scheme

As mentioned in the discussion section, the ultimate solution would probably be some kind of multi-party computation (mpc). Although we are not attempting to solve that problem today, we can still contemplate more advanced schemes. The CashFusion scheme attempts to shard the information between the server and all the players, but it could be improved by relying less on the server. This may not have a large practical impact, but it is a good area of research and may help create more distributed solutions for the future.

What follows is a hybrid scheme that combines an mpc component with a similar structure to CashFusion:

Instead of sharing lists of component commitments with the server, the component commitments would also be announced covertly so that initially, no one knows which player owns a particular commitment.

For each commitment, a random subset of players (including the actual owner) each produce a random number and join a round of multiparty computation to sum their secrets, and commit to the numbers under a homomorphic encryption scheme. A player will choose an even number if he is not the owner, and an odd number if he is.

The normal fusion verification process is performed, and when a faulty component commitment is found, the non-owner participants of the mpc reveal their commitments along with the homomorphic sum. By simple subtraction, the owner is revealed to have commited to an odd number.

The dishonest disrupter can attempt to avoid this by not participating in some rounds or falsely submitting the wrong information, but then these commitments will be retried and anti-DOS countermeasures taken, perhaps including another homomorphic operation showing each player committed to ownership of some number of inputs.

In addition, the Pedersen commitments used in CashFusion need to be unbinded from the component commitments. Instead, the players submit a zero-knowledge proof (ZKP) for each amount that shows the value was commited to in one of the Pedersen commitments, without revealing which commitment. $C(a) = x, x \in X$, where a is the amount, and X is the set of all players' commitments. This requires all inputs and outputs to be unique amounts.