Byzantine Cycle Mode: Scalable Bitcoin Mixing on Unequal Inputs

sundance*

August 23, 2014

Abstract

We present a new distributed algorithm, called *Byzantine Cycle Mode* (BCM), that mixes bitcoins of different sizes. Most known decentralized, risk-less bitcoin mixing algorithms (CoinSwap[4], CoinShuffle[5], CoinShift[7], Dark Wallet's CoinJoin[1]) either require the numbers of bitcoin being mixed to be equal or their anonymity strongly depends on it. Some also do not scale easily to large number of players. BCM relaxes these constraints by transforming large instances with unequal bitcoin amounts into smaller sub-instances of equal amounts — allowing players to mix using the known algorithms while preserving their degree of semantic security.

1 Introduction

Suppose N players $i \in \mathcal{N} = \{1, 2, \ldots, N\}$ want to mix an (integer) number of satoshis, respectively $\{n_i | i \in \mathcal{N}\}$. Most known decentralized bitcoin mixing algorithms, however, either require the numbers of bitcoin being mixed to be equal or their anonymity strongly depends on that being the case. For example, CoinSwap, CoinShuffle, and CoinShift assume equal bitcoin amounts. CoinJoin is subject to subset-sum attacks if the amounts are different.

As Bitcoin transactions exist on the blockchain in the clear for all to see indefinitely, we should not under estimate an adversary's will and means to de-anonymize its target's transactions. It is necessary to explore mixing techniques

*email: sundance30203@gmail.com

that allow us to scale to large numbers of players, as well as create information-theoretic barriers (as opposed to only computational barriers) to an adversary's network transaction analysis. We propose that scalable anonymity solutions require not only cryptographic techniques but also techniques from areas such as distributed algorithms, combinatorial optimization, and graph theory.

We present a new algorithm called *Byzantine Cycle Mode* (BCM) to address this. BCM's name derives from its characteristics:

- **Byzantine** BCM uses a Byzantine agreement process, where the output is based on randomized contributions from independent players. It tolerates honest failures from less than 1/2 of the players, and allows the players to abort or boot out persistently bad players.

- **Cycle** BCM decomposes the output into sets of edge-disjoint network graph cycles. Only the set of players in each cycle participate in the mix execution on the blockchain.

- **Mode** The algorithm is called a mode because its relationship to the underlying mixing primitives (CoinJoin, CoinShuffle, CoinShift, etc) resembles that of encryption modes (such as counter mode and CBC) to underlying block ciphers. In particular, encryption modes extend the application of block ciphers like AES and 3DES to large inputs which do not necessarily fit along the block size boundaries. Analogously, BCM extends the application of mixing primitives to large numbers of players who are mixing unequal sizes.

1.1 Key Contributions

The key contribution of this work is a mixing process that produces more varied and complex mixings and that takes a wider range of inputs than that of any of the mixing primitives.

More precisely, if Bitcoin mixers are considered to be pseudo-random functions (PRFs) mapping bitcoin amounts from players' input addresses to their output addresses[1], then the key contribution of this work can be characterized as a PRF with the following properties:

1. Has (a) larger domain and range than the mixing primitive, or (b) better semantic security on the domain/range than the mixing primitive.

[1]In the case of coinjoins, consider the satoshis uniformly distributed.

2. Can not be disproportionately influenced by an active adversary.

3. Preserves the semantic security of the underlying mixing primitive (ie, no information leaks before the primitive is executed).

4. Is generic and suits any underlying mixing primitive.

1.2 Assumptions and Conventions

We rely on the following assumptions:

1. There exists a mutually understood ordering on the players, presumably provided during peer discovery (which is outside the scope of this work), such that the players are numbered $1, 2, ..., N$. The choice of ordering is insignificant.

2. All the players have pairwise network connectivity to each other over which authenticated-encryption-secured, reliable messaging may be done.

3. IP addresses used in the connections are obscured through onion routing.

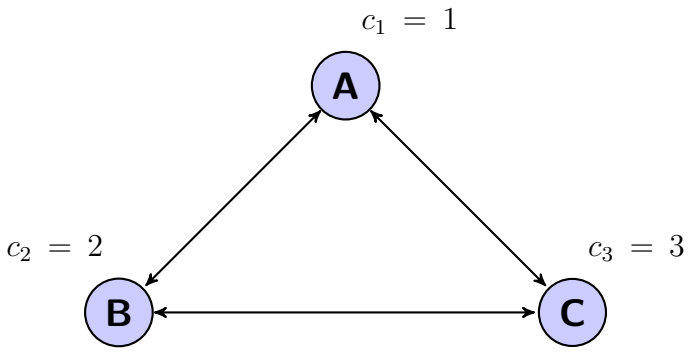
4. The security assumptions of block ciphers, collision-resistant hash functions, and public-key cryptography are valid.

Unless otherwise noted, we use the term *broadcast* to mean that a player sends a message to each of his peers — as opposed to its common Bitcoin usage for denoting propagation of Bitcoin transactions.

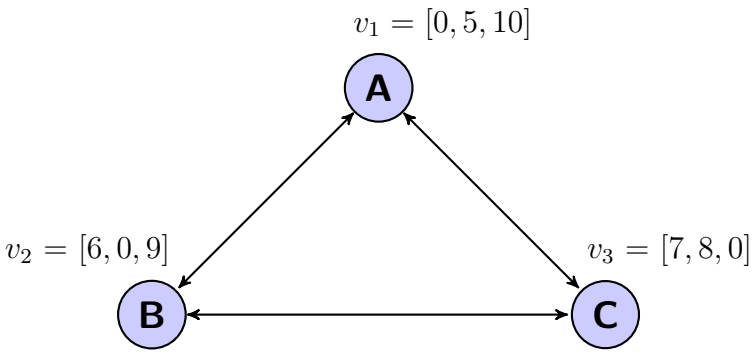
While the algorithms in this paper use integer representations of bitcoin amounts (as numbers of satoshis), the examples however give amounts in numbers of bitcoin, for the sake of presentation. For example, instead of 100,000,000, the number 1 is written.

2 Overview

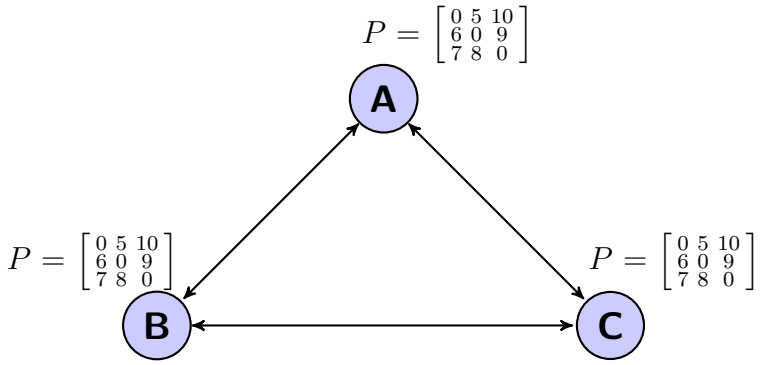
Let's suppose that Alice, Bob, and Charlie want to mix $c_1 = 1$, $c_2 = 2$, and $c_3 = 3$ bitcoins, respectively. After peer discovery, they establish with each other pairwise, secure network connections which are separate from those managed by their Bitcoin clients. The network graph is shown below.



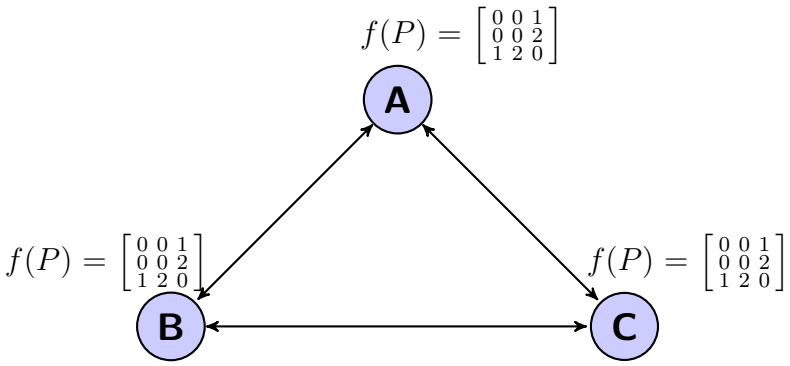
Step 1. Each player generates random inputs and broadcasts the inputs to each peer.



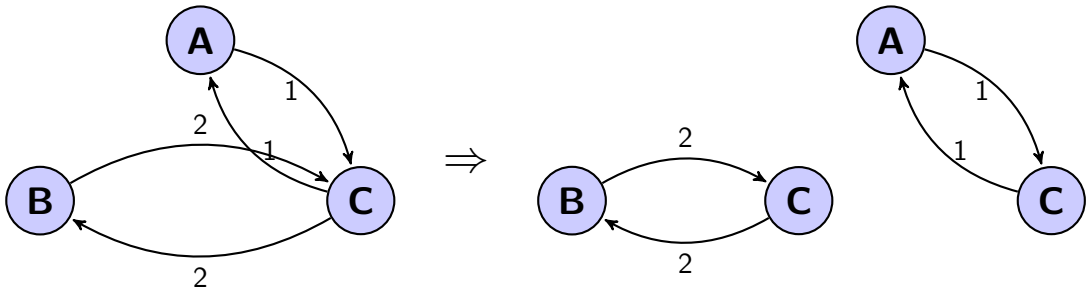
Step 2. Each player receives all of the inputs.



Step 3. Using Byzantine agreement, the players agree on the output of a function f, giving the bitcoin flow adjacency matrix.



Step 4. Finally, the players execute mixes (as determined by f) on the Bitcoin blockchain using known decentralized mixing techniques such as CoinJoin, CoinSwap, CoinShuffle, or CoinShift. In this case, players A and C mix 1 BTC, and players B and C mix 2 BTC using one of the mix algorithms.



3 The Mix Agreement Protocol

In this section we give the Byzantine Agreement protocol that allows the honest players to agree on how many coins each player will send to other players. The agreement is done in a way such that no one player can disproportionately influence the output and all players' inputs influence the output.

The Byzantine Agreement model uses is the following assumptions:

- **Connected** — each player has network connectivity to each other player
- **Synchronized** — the protocol is executed in a sequence of rounds. Each round includes three steps from each player: receive a message from each player, perform a calculation, broadcast a response to all players
- **Broadcast** — messages are broadcast to all peers and in a broadcast, an honest player sends the same message to all peers.
- **Active/adaptive adversary** — an adversary may control a number of players, receive the messages they receive, and adapt its behavior based on the messages received. Because all honest players send the same messages to all players, the adversary is assumed to know all of the messages sent to any players.
- **Reliable links** — the adversary may not control, reorder, drop, or change messages between players. This strong assumption is admissible because the mix agreement protocol is designed to be implemented in conjunction with a

transport layer that supports reliable, ordered delivery with fast disconnect detection through heartbeats, and application-level replay capability.

3.1 The Mix Allocation Matrix

A *mix allocation* (informally, a *mix*) is an $N \times N$ integer matrix $[x_{ij}]_{i,j \in \mathcal{N}}$ such that x_{ij} gives the number of satoshis that player i should transfer to player j as part of the mixing process. The total number of satoshis each player sends to his peers must equal that which he receives. Thus for all $i \in \mathcal{N}$, the ith row and ith column sum to the same value. Formally:

Definition 3.1 (Mix allocation)**.** *A mix allocation for* $[n_1, n_2, \ldots, n_N]$ *is an* $N \times N$ *integer matrix* $[x_{ij}]$ *such that*

$$\forall i, \sum_{j=1}^{N} x_{ij} = n_i \tag{1}$$

$$\forall j, \sum_{i=1}^{N} x_{ij} = n_j \tag{2}$$

□

For example, three mix allocation matrices are given below. Note that mix allocation matrices need not be symmetric, and they also are not unique; when $N > 1$, there are multiple possible mix allocation matrices.

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 2 \\ 1 & 2 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 2 \\ 0 & 3 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 1 & 3 & 0 \end{bmatrix}$$

(a) a mix for $[1, 2, 3]$ (b) a mix for $[1, 3, 3]$ (c) a mix for $[1, 2, 3, 4]$

Figure 1: Three example mix allocation matrices

A mix allocation matrix defines a directed transaction graph of bitcoins between players, where the graph nodes are the players and the edges are the bitcoin amounts being transferred, either through a join or swap. The properties of these

graphs are used in this work (see section 3.6) to decompose the task of mixing according to the allocation matrix into smaller, independent mixing tasks to be executed by the players.

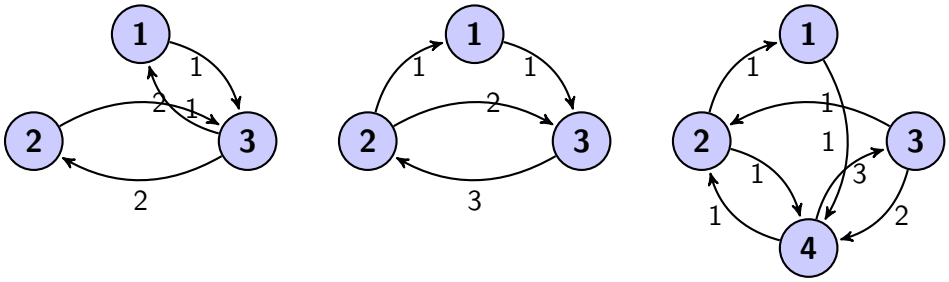


Figure 2: Corresponding mix graphs for the three matrices in figure 1, respectively

3.2 Protocol Description

The result of the Mix Agreement protocol is that all honest players either agree on a mix allocation or they all abort the process. The below simplified protocol is run simultaneously by all players after player discovery is completed and secure network connections are established.

At the termination of this protocol, if agreement is achieved, then each player uses this mix allocation to execute the mix using a decentralized mix algorithm such as CoinJoin, CoinShuffle, or CoinShift.

Algorithm 1: Mix Allocation Agreement Protocol

input : i, player's index
input : N, number of players
begin
  `ExchangeInputs` (i, N)
  $M_i(1) \leftarrow$ `ComputeMixAllocation`$([v_1, v_2, \ldots, v_N])$;
  for *rounds* $s = 1, 2, 3$ **do**
    broadcast $M_i(s)$;
    for $\forall j \neq i$ **do**
      receive $M_j(s)$ from player j;
      if $\exists M =$ *unique, most frequent value amongst* $\{M_j(s)\}_{j \neq i}$ **then**
        $M_i(s+1) \leftarrow M$
      end
      else abort;
      $c(s) \leftarrow$ count of M;
      if $c(s) \leq \frac{1}{2}N$ **then** abort;
      if $c(s) > \frac{2}{3}N$ **then** assume agreement on M;
    end
  end
  if *no agreement* **then** abort;
  else `ComputeCycles` ();
end

Note that agreement is not guaranteed from this protocol; any number of players may abort before agreement is achieved. As such, it is not a solution to the Byzantine Generals Problem, proper. This relaxation is chosen in the context of bitcoin mixing, because it is more desirable to allow a player to abort (thereby mitigating the effectiveness of denial-of-service tactics) than it is to necessarily achieve agreement. Once a player aborts, he can subsequently retry another mix agreement with presumably different players. No bitcoins or sensitive information are transferred during mix allocation agreement, hence there are little to no abort costs or risks aside from the nominal computation time to execute the protocol.

3.3 Mix Allocation Computation

One technique for achieving Byzantine Agreement is to define a function that, when executed by honest players on inputs from all players, deterministically returns the same value for each honest player.[2] The key contribution of this work is the definition of such a function, ComputeMixAllocation(), for bitcoin mixes.

ComputeMixAllocation() is a deterministic, well-defined procedure that takes an $N \times N$ matrix and returns a canonical mix allocation based on its inputs. It is designed to return the same result for all honest players and to be robust to factors such as different implementation language or difference in computing environment/architecture.

Algorithm 2: ComputeMixAllocation

input : $P = [p_{ij}]$, $N \times N$ allocation priority matrix
output: $M = [x_{ij}]$, $N \times N$ allocation matrix

begin
  pairs $\leftarrow (i,j)_{i,j \in \{1 \ldots N\}^2}$; `// list of cell positions of` P
  inventory $\leftarrow [n_1, n_2, \ldots, n_N]$; `// num coins available`
  capacity $\leftarrow [n_1, n_2, \ldots, n_N]$; `// num coins needed`
  sort pairs according to non-decreasing i;
  stable sort pairs according to non-increasing p_{ij};
  for $(i,j) \in$ pairs *in order* **do**
    $x_{ij} \leftarrow min(\text{capacity}_j, \text{inventory}_i)$;
    $\text{inventory}_i \leftarrow \text{inventory}_i - x_{ij}$;
    $\text{capacity}_j \leftarrow \text{capacity}_j - x_{ij}$;
  end
  return $M = [x_{ij}]$
end

ComputeMixAllocation() has the following properties:

1. The function runs in time $\mathcal{O}(N^2 \log N^2)$, due to the sorting of N^2 matrix cell positions.

2. The output matrix M has at most $2N$ non-zero entries. This is because for

each iteration in the **f**or loop, at least one of capacity$_j$ or inventory$_i$ gets annihilated, and this can happen at most $2N$ times.

3. The output is well-defined and robust in terms of consistency. As simple integer arithmetic only is needed, it is reasonable to expect that the same value will be returned on different execution platforms.

Below is an execution trace of ComputeMixAllocation() on $[1, 2, 3]$ and $P = \begin{bmatrix} 0 & 5 & 10 \\ 6 & 0 & 9 \\ 7 & 8 & 0 \end{bmatrix}$. Since the result is reached by the end of iteration four, iterations five through eight are omitted.

iteration	(i, j)	M	*inventory*	*capacity*
0		$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	$[1, 2, 3]$	$[1, 2, 3]$
1	$(1, 3)$	$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	$[0, 2, 3]$	$[1, 2, 2]$
2	$(2, 3)$	$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 2 \\ 0 & 0 & 0 \end{bmatrix}$	$[0, 0, 3]$	$[1, 2, 0]$
3	$(3, 2)$	$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 2 \\ 0 & 2 & 0 \end{bmatrix}$	$[0, 0, 1]$	$[1, 0, 0]$
4	$(3, 1)$	$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 2 \\ 1 & 2 & 0 \end{bmatrix}$	$[0, 0, 0]$	$[0, 0, 0]$

Figure 3: Execution trace of ComputeMixAllocation().

3.4 Input Exchange

ExchangeInputs() is a straight forward broadcast of the players' inputs, with an additional hash commitment and verification. Each player first broadcasts a hash of the player's inputs and number of coins he desires to mix before broadcasting those values. This enforces that all players commit their inputs before learning any more information from the protocol about other player. The hash function used should be a collision-resistant one such as SHA-256.

Algorithm 3: ExchangeInputs

```
   input  : i, player's index
   input  : c_i, num of satoshis player i wants to mix
   input  : N, number of players
 1 begin
 2     v_i ← GenerateInputs(i, N) ;
 3     send and receive input commitments;
 4     broadcast Hash(c_i, v_i) ;
 5     for ∀j ≠ i do
 6         receive Hash(c_j, v_j) from player j;
 7         commit_j ← Hash(c_j, v_j);
 8     end
 9     send, receive, and verify inputs;
10     broadcast c_i, v_i ;
11     for ∀j ≠ i do
12         receive v_j from player j;
13         if Hash(c_j, v_j) ≠ commit_j then
14             abort;
15         end
16     end
17 end
```

3.5 Input Generation

GenerateInputs() is a procedure that produces the player i's input contribution. It returns an integer vector $v_i = [v_{i1}, v_{i2}, \ldots, v_{iN}]$ such that

$$\sum_{j=1}^{N} v_{ij} = C \tag{3}$$

$$v_{ii} = 0 \tag{4}$$

where C is an apriori known constant. Once the inputs vectors are exchanged, the players form an $N \times N$ matrix $P = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_N \end{bmatrix}$ that is the input to ComputeMixAllocation(). P represents a prioritization of which pairs ComputeMixAllocation() tries to match.

The significance of (3) is that it forces a trade-off for an active adversary attempting to control the outcome. Further, the fact that the inputs are generated in the blind means that an adaptive adversary does not have the opportunity to tailor his inputs based on input of his peers. The significance of (4) is that ComputeMixAllocation() prioritizes mixing distinct players' bitcoins rather than a player with himself.

The input vector, ideally, is generated uniformly over the space satisfying the above using a pseudo-random number generator. The quality of randomness in a player's contribution is at the discretion of the player. That said, the honest player is incentivized to use a good pseudo-random number generator, since doing so reduces the advantage a dishonest player has in knowing how their inputs will interact.

The constant C should be an integer sufficiently large such that the expected number of collisions in inputs is low. This ensures that the prioritization is determined primarily by the inputs, chosen by the players, rather than the player ordering, chosen somehow during peer discovery. By the Birthday Paradox analysis, $C = 65,536$ is sufficient for mixes for N up to a several hundred players. There is no cost in choosing a higher value for C.

3.6 Decomposition Into Cycles

The mix allocation matrices returned by ComputeMixAllocation() have equivalent representations as directed transaction graphs. Due to constraints (1) and (2), these graphs have a property known as flow conservation.

Definition 3.2 (Flow Conservation)**.** *Flow conservation is the property that for all vertices, the sum of the weights flowing into the vertex is equal to the sum of the weights flowing out of the vertex.*

Given a graph with flow conservation properties, the graph can be decomposed into simple cycles which also exhibit the flow conservation property.[6] ComputeCycles() does this using Johnson's well known graph algorithm. Because mix allocation matrices have at most $2N$ non-zero entries, the running time is $\mathcal{O}(N^2)$. For example, the following graphs are decomposed into simple cycles.

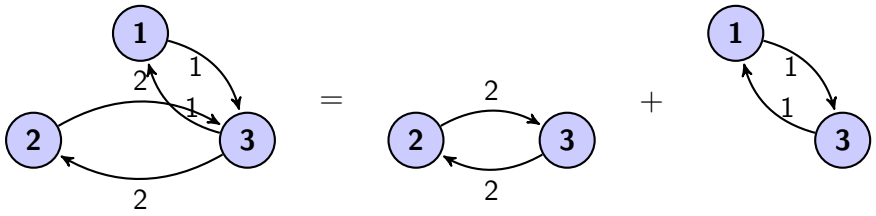


Figure 4: Decomposition of 1(a) transaction graph.

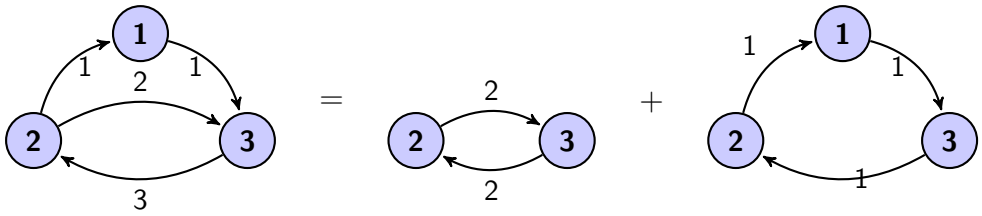


Figure 5: Decomposition of 1(b) transaction graph.

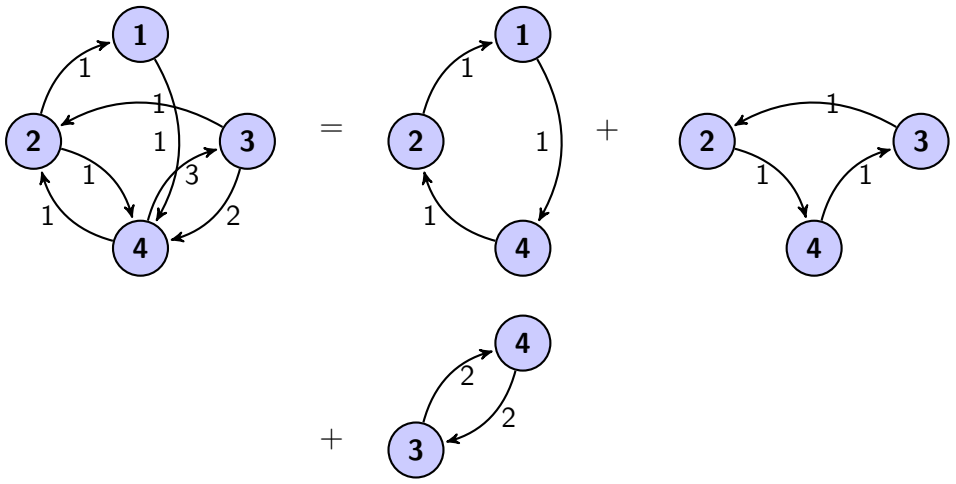


Figure 6: Decomposition of 1(c) transaction graph.

Each of the simple cycles on the right hand side represent separate mixes to be executed by the nodes of that cycle. The mix amount is the weight that is in common between the edges of the cycle. These mixes can be executed independently of each other, and because of the flow conservation property of each cycle, the success or failure of any given cycle's mix does not pose a risk exposure to any players with respect to the other mixes.

4 Open Questions

1. Can a reasonable, formal definition of semantic security be formulated for bitcoin mixing?

2. Can any mixing technique (as opposed to ZeroCoin's sidechain/cryptographic technique) meet a reasonable definition of semantic security?

3. What other modes are definable? For instance, interpreting graphs with nested cycles into nested coinjoins, where one output of a join is a direct input to another join.

4. Is there a characterization of the parameters (number of players, input amounts) for which either CoinShuffle, CoinJoin, or CoinShift provides better semantic security than the others?

5 Appendix: Optimal (Centralized) Bitcoin Mixing

The construction of the ComputeMixAllocation() algorithm was inspired by a solution to an integer combinatorial optimization problem, called CoinMixOpt, that arises for a centralized bitcoin mixing service. This solution was the original focus of this work, and we later discovered that it has relevance in the decentralized context as well. It is worth digressing into for the benefit of the reader. An algorithm solving CoinMixOpt enables a centralized mixing service to compute and execute an optimal mixing on behalf of its clients without the use of reserve coins.

The aim of CoinMixOpt is to find a mix $M = [x_{ij}]$ such that the total taint of the mixed coins (should the mix happen) is minimized. Let the $N \times N$ matrix $[\lambda_{ij}]$ be the correlation (taint) matrix, calculated to a constant depth of transactions, and let $P = [p_{ij}] = [1 - \lambda_{ij}]$ be a profit matrix. CoinMixOpt can be stated as the following profit maximization problem.

$$\text{maximize} \sum_{i=1}^{N} \sum_{j=1}^{N} p_{ij} x_{ij}$$

$$\text{subject to}$$

$$\sum_{j=1}^{N} x_{ij} = n_i$$

$$\sum_{i=1}^{N} x_{ij} = n_j$$

By maximizing the profit, the overall taint is minimized, as desired.

CoinMixOpt is a special case of the Generalized Assignment Problem (GAP), an integer optimization problem that is **NP**-hard. GAP also is **APX**-hard — which means that unless $\mathbf{P} = \mathbf{NP}$, their exists no polynomial time algorithm that approximates the optimal solution for GAP to within a constant factor. Fortunately, CoinMixOpt on the other hand is in **P**, and it is solved in $\mathcal{O}(N^2 \log N^2)$ time with the following adaptation of the greedy approximation algorithm for GAP.

Algorithm 4: GreedyOpt

input : $P = [p_{ij}]$, $N \times N$ profit matrix
output: $M = [x_{ij}]$, $N \times N$ allocation matrix

begin
  pairs $\leftarrow (i,j)_{i,j \in \{1 \ldots N\}^2}$; `// list of cell positions of` P
  inventory $\leftarrow [n_1, n_2, \ldots, n_N]$; `// num coins available`
  capacity $\leftarrow [n_1, n_2, \ldots, n_N]$; `// num coins needed`
  sort pairs according to non-increasing p_{ij};
  for $(i,j) \in$ pairs *in order* **do**
    $x_{ij} \leftarrow min(\text{capacity}_j, \text{inventory}_i)$;
    $\text{inventory}_i \leftarrow \text{inventory}_i - x_{ij}$;
    $\text{capacity}_j \leftarrow \text{capacity}_j - x_{ij}$;
  end
  return $M = [x_{ij}]$
end

Aside from a minor change in the sorting, ComputeMixAllocation() is identical to GreedyOpt(). ComputeMixAllocation(), however, aims for consistent output amongst independent players, rather than optimality.

References

[1] DarkWallet. CoinMixing. `https://wiki.unsystem.net/en/index.php/DarkWallet/CoinMixing`, 2013.

[2] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[3] Greg Maxwell. CoinJoin: Bitcoin privacy for the real world. `https://bitcointalk.org/index.php?topic=279249.0`, 2013.

[4] Greg Maxwell. CoinSwap: Transaction graph disjoint trustless trading. `https://bitcointalk.org/index.php?topic=321228.0`, 2013.

[5] Tim Ruffing, Pedro Moreno-Sanchez, and Aniket Kate. CoinShuffle: Practical decentralized coin mixing for Bitcoin. In *ESORICS'14*, volume 8713 of *Lecture Notes in Computer Science*, pages 345–364. Springer, 2014.

[6] L. Sun and M. Wang. An algorithm for a decomposition of weighted digraphs — with applications to life cycle analysis in ecology. *Journal of Mathematical Biology*, 54:199–226, 2007.

[7] sundance. CoinShift: Atomic n-way coin swapping. Forthcoming, 2014.