Byzantine Cycle Mode: Scalable Bitcoin Mixing on Unequal Inputs

No Author Given

No Institute Given

Abstract. We present a new distributed algorithm, *Byzantine Cycle Mode* (BCM), that mixes bitcoin inputs of different sizes. The known decentralized risk-less bitcoin mixing algorithms either assume equal inputs for operation or become vulnerable if the inputs are unequal. BCM relaxes this constraint by transforming instances with unequal bitcoin amounts into smaller sub-instances of equal amounts.

1 Introduction

The invention of Bitcoin has brought new forms of financial practices, actors, and services. There is often sensitivity (in the form of operational risks and risks to client confidentiality and personal financial privacy) related to the information that is both directly available on the blockchain and deducible through transaction graph analysis by unrelated third parties (adversaries). The many examples include the following:

- Bitcoin exchanges, payment processors, and some online wallets have custodial responsibilities, whereby they receive control of bitcoins from clients and are responsible for holding or performing actions on behalf of the client.
- Companies that pay its employees in bitcoins will want to obscure the amounts given to particular employees.
- Political and membership-based organizations may want to keep their membership lists private while receiving fees/donations from members' Bitcoin addresses.

The Bitcoin protocol, aptly, neither hinders nor directly supports this type of transactional obfuscation. Additional technical mechanisms are needed in order to give these new practices the necessary confidentiality protection. Bitcoin mixing is a technique introduced by the community to mechanize such protection with the aim to render deductible information

less useful and/or more expensive to obtain. This work aims to build on this approach by introducing a new mixing algorithm based on existing Bitcoin functionality.

This work aims to make it easier for actors to engage in mixing with large anonymity sets, where the input amounts are not apriori known or controlled. The known decentralized bitcoin mixing algorithms either require the input amounts to be equal or their anonymity strongly depends on that being the case. For example, CoinSwap [[4]] and CoinShuffle [[5]] assume equal bitcoin amounts. CoinJoin [[1]] is subject to subset-sum attacks if the amounts are different. It should be noted that transactions resulting from mixing operations *are distinguishable* from regular transactions (with non-negligible advantage), despite the historical aim to mask mix transactions as regular transactions. The size of the anonymity set of a mix operation is therefore limited by the number of addresses *that participate in the mixing operation* and not the total number of addresses that are active on the Bitcoin network at the time of the operation. Therefore, the effectiveness of mixing depends on our ability to perform mixes on large numbers of addresses. We present an algorithm that facilitates large-scale mixing on disparate inputs amounts so as to allow actors to engage more readily with large anonymity sets than previously possible.

Suppose N players $1, 2, \ldots, N$ want to mix an (integer) number of satoshis, $\{n_i \,|\, i \in \mathcal{N} = \{1, 2, \ldots, N\}\}$, respectively. The players must jointly construct and agree on a mix specification and execute the mix transactions under the following conditions:

1. The algorithm is distributed and decentralized.
2. There is zero risk on any player's part of losing coins.
3. There are no trusted third parties involved.
4. There are no additional mixing fees.
5. Only well-known cryptographic techniques are used.
6. The algorithm must scale to a large number of players.
7. No one player may disproportionately influence the mix agreement.
8. All honest players detect Byzantine faults and identify their source thereby allowing them to exclude the faulty player(s) from the final mix.

The key contributions of this work are a new mixing algorithm satisfying the above conditions, and a new method for detecting Byzantine

faults in the mix agreement and identifying their source. We present a new algorithm called *Byzantine Cycle Mode* (BCM), the name of which derives from its characteristics:

- **Byzantine.** BCM uses a Byzantine agreement process to construct a mix, where the output is based on randomized contributions from independent players. It tolerates Byzantine failures from up to 1/2 of the players, and allows the players to identify the source of the failures.
- **Cycle.** BCM decomposes the mix agreement output into sets of edge-disjoint network graph cycles, each of which define the set of players that will participate in the corresponding mix execution on the blockchain.
- **Mode.** The algorithm is called a mode because its relationship to the underlying mixing primitives (CoinJoin, CoinShuffle, etc) resembles that of encryption modes (such as CTR and CBC) to underlying block ciphers. In the same way that encryption modes (e.g., GCM, EAX) extend the application of block ciphers (e.g., AES, 3DES) to large inputs which do not necessarily fit along the block size boundaries, BCM extends the application of mixing primitives to large numbers of users who are mixing unequal sizes.

Our threat model and computational model include the following assumptions:

1. **Player ordering.** 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. **Connected.** All the players have pairwise network connectivity to each other over which reliable messaging with authenticated encryption may be done.
3. **Secure messaging.** Players reveal their public keys during peer discovery, and all messaging is encrypted using secure ciphers and public-key encryption.
4. **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.

5. **Broadcast.** Messages are broadcast to all peers and in a broadcast, an honest player sends the same message to all peers.
6. **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.
7. **Fallibility and DOS resistance.** In the presence of Byzantine faults, any player may opt to abort the procedure, and if the faulty process is identifiable amongst the participants then players may exclude the faulty process from further steps.
8. **Reliable links.** The adversary may not control, reorder, drop, or change messages between players. This strong assumption is admissible because the mix agreement protocol was designed in tandem with an ongoing software implementation project that uses a transport layer that supports reliable, ordered delivery with fast disconnect detection through heartbeats, and application-level replay capability.
9. **IP address privacy.** The IP addresses used in the network connections are obscured through onion routing.
10. **Cryptography semantic security.** The security assumptions of one-way functions, collision-resistant hash functions, and public-key cryptography are valid.
11. There are greater than $N/2$ honest players.

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.

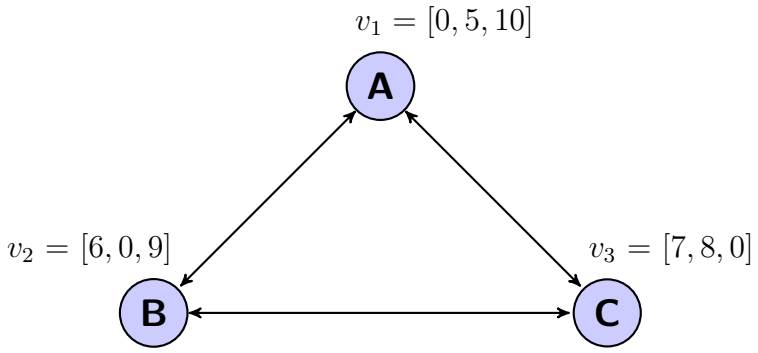
Unless otherwise noted, the players used the following public-key messaging format — where the cypher-text is appended by the public-key encryption of the symmetric key using each player's public key. Let m be a plain text message. Let (E_1, D_1) be a secure symmetric encryption scheme with MAC authentication of associated data, such as AES-128 in GCM mode. Let (E_2, D_2) be a secure public-key encryption scheme (such as RSA or El Gamal), and (pk_i, sk_i) be the set of public and private keys

for each player. To broadcast, a player randomly chooses a symmetric key k and sends $E_1(k, m) \,||\, pk_i \,||\, E_2(pk_1, k) \,||\, pk_2 \,||\, E_2(pk_2, k) \,||\, \ldots \,||\, pk_N \,||\, E_2(pk_N, k)$ to all of its peers, where $||$ denotes concatenation. To decrypt a received broadcast, the player i finds its public key in the cypher-text, and computes $D_1(D_2(sk_i, E_2(pk_i, k)), E_1(k, m))$.

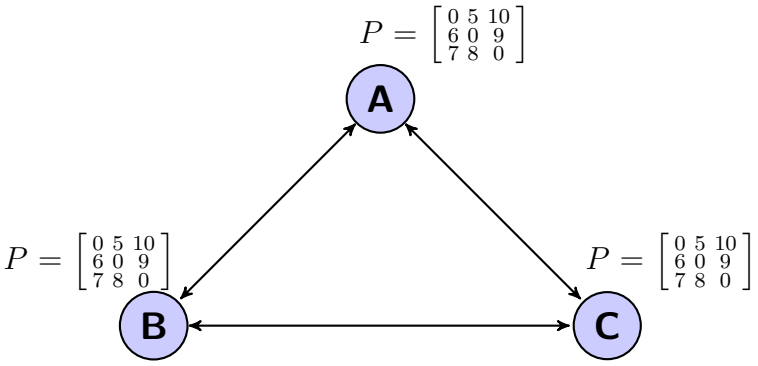
2 Overview with an Example

Let's suppose that Alice, Bob, and Charlie intend to mix $c_1 = 1$, $c_2 = 2$, and $c_3 = 3$ bitcoins, respectively.

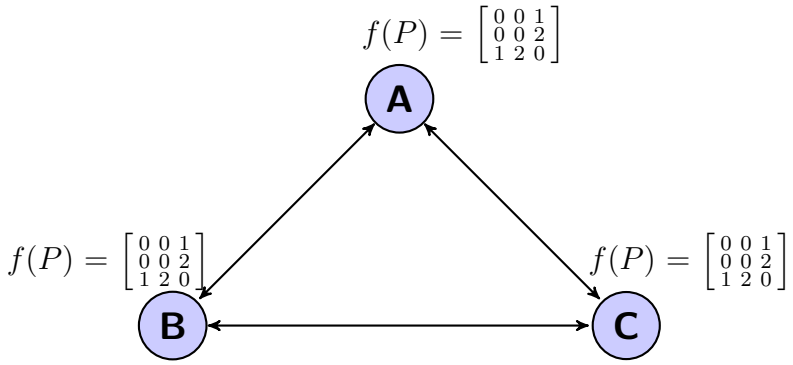
Step 1. Each player generates random inputs and broadcasts the inputs to each peer.



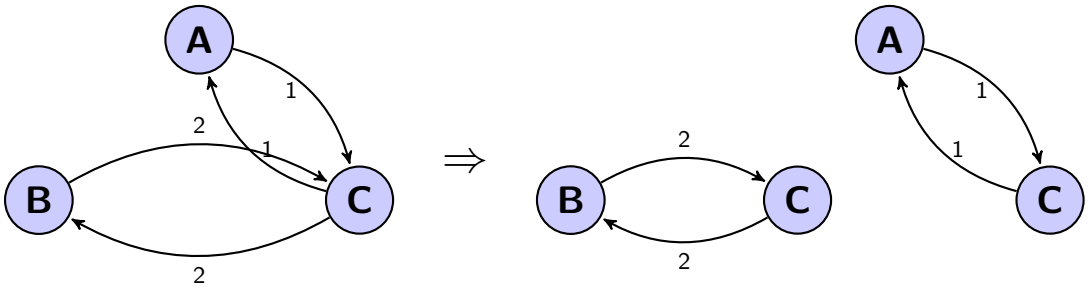
Step 2. Each player receives and verifies inputs from the other players.



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 or CoinShuffle.



Let's suppose that CoinShuffle is used. Then, players B and C perform a CoinShuffle on 2 BTC, while players A and C independently perform a CoinShuffle on 1 BTC.

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. 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 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 $[1, 2, 3, 4]$

Fig. 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 3.5) to decompose the task of mixing according to the allocation matrix into smaller, independent mixing tasks to be executed by the players.

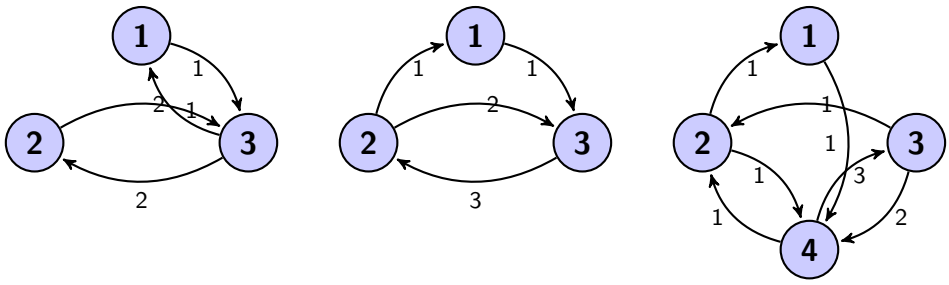


Fig. 2: Corresponding mix graphs for the three matrices in figure 1, respectively

3.1 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 or CoinShuffle.

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` (M);
end

Note that agreement is not guaranteed from this protocol; any player may abort before agreement is achieved to allow protection from denial-of-service attacks. As such, it is not a solution to the Byzantine Generals

Problem. After aborting, a player can subsequently retry another mix agreement with presumably different players.

3.2 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 × N allocation priority matrix
   output: M = [x_ij], N × N allocation matrix
 1 begin
 2 |  pairs ← {(i, j) | i, j ∈ 𝒩}; // list of cell positions of P
 3 |  inventory ← [n_1, n_2, ..., n_N];        // num coins available
 4 |  capacity ← [n_1, n_2, ..., n_N];            // num coins needed
 5 |  sort pairs according to non-decreasing i;
 6 |  stable sort pairs according to non-increasing p_ij;
 7 |  for (i, j) ∈ pairs in order do
 8 |  |  x_ij ← min(capacity_j, inventory_i);
 9 |  |  inventory_i ← inventory_i − x_ij;
10 |  |  capacity_j ← capacity_j − x_ij;
11 |  end
12 |  return M = [x_ij]
13 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 **for** 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.

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	$(3, 1)$	$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 2 \\ 1 & 2 & 0 \end{bmatrix}$	$[0, 0, 0]$	$[0, 0, 0]$

Fig. 3: Execution trace of ComputeMixAllocation().

3.3 Input Exchange

The algorithm ExchangeInputs exchanges inputs between the players and allows each honest player to detect Byzantine faults during the inputs

exchange. Authenticated Byzantine agreement is used on the initial hash value exchange to protect against denial of service attacks, and each player first broadcasts a hash of his inputs to ensure that all players commit their inputs before learning the inputs of other players.

The input exchange algorithm uses a two-step digitally signed broadcast scheme. In the first step, each player broadcasts his input hash commitment value with a digital signature of the encrypted message. In the second step, for each signed message received in step 1 that validates against the sender's signature, each player relays (re-broadcasts) the message, digitally signed by its private key. If the original message did not validate against the sender's signature, the relay broadcasts an null message.

The network reliability, broadcast, and secure communication assumptions allow each player to validate whether the sender sent the same message to all peers. Each player performs the following validations in order:

1. The relayed message must validate against the relay's public key. Otherwise, either the relay is at fault or there is a man-in-the-middle attempt. The receiving player may abort or attempt to exclude the relay.
2. The original message must validate against the sender's public key. Otherwise, the relay can be faulted; it should relay if and only if this validation succeeded.
3. During decryption, the original message must MAC authenticate against the symmetric key. Otherwise, the sender can be faulted; the sender's digital signature validated, so no third party could have broken the MAC authentication of an otherwise proper cypher-text.
4. There should be a single, most frequent decrypted original message (hash input commitment). Otherwise, the sender can be faulted. Given that there are greater than $N/2$ honest players, the cypher-texts for their relayed messages honestly reflect what the sender sent to them. Therefore, if there is no most frequent original message, then it can only be because the sender varied a sufficient number of its original messages amongst the other players or sent a bad digital signature.
5. The subsequently sent input values should hash to the hash commitment input value. Otherwise, the sender is faulted, clearly for the same reasoning above.

6. The subsequently sent input values should meet the constraints 3, 4, and 5 given in the next section. Likewise, the sender is faulted if this is not the case.

Let pk_i be the public key of player i, sk_i be the secret key of player i, H be a collision resistant hash function, and (S, V) be a secure public-key digital signature scheme.

Algorithm 3: ExchangeInputs

input : i, player's index; c_i, number of satoshis player i intends to mix; N, number of players

begin
  $v_i \leftarrow \texttt{GenerateInputs}(i, N)$;
  $m_i \leftarrow \texttt{H}(c_i, v_i)$;
  broadcast $m_i \,||\, \texttt{S}(sk_i, m_i)$;
  for $\forall j \neq i$ **do**
    receive $m_j \,||\, \texttt{S}(sk_j, m_j)$ from player j;
    if $V(pk_j, m_j) = 1$ **then**
      $m'_i \leftarrow m_j \,||\, \texttt{S}(sk_j, m_j)$;
      broadcast $m'_i \,||\, \texttt{S}(sk_i, m'_i)$;
    end
    $Q_j = \varnothing$;
  end
  for $\forall j, k \neq i$ **do**
    receive $m_j \,||\, \texttt{S}(sk_j, m_j) \,||\, \texttt{S}(sk_k, m'_j)$;
    if $V(pk_k, m'_j) = 1$ **then** add m_j to Q_j;
  end
  for $\forall j \neq i$ **do**
    if $\exists$ *unique, most frequent value* m *in* Q_j **then** $\mathsf{commit}_j \leftarrow m$;
  end
  broadcast c_i, v_i;
  for $\forall j \neq i$ **do**
    receive v_j from player j;
    if $\texttt{H}(c_j, v_j) \neq \mathsf{commit}_j$ **then** abort;
  end
end

3.4 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

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

$$v_{ij} \neq 0, \text{ if } i \neq j \tag{4}$$

$$v_{ii} = 0 \tag{5}$$

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 (5) 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.

3.5 Decomposition Into Cycles

The mix allocation matrices returned by ComputeMixAllocation() have equivalent representations as directed transaction graphs. Due to con-

straints (1) and (2), these graphs have a property known as flow conservation.

Definition 2 (Flow Conservation) *Flow conservation is the property that for all vertexes, the sum of the weights flowing into the vertex is equal to the sum of the weights flowing out of the vertex.*

Any graph that has the flow conservation property can be decomposed into simple cycles that also have 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.

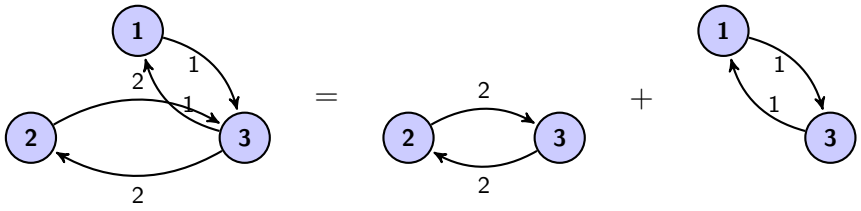


Fig. 4: Decomposition of 1(a) transaction graph.

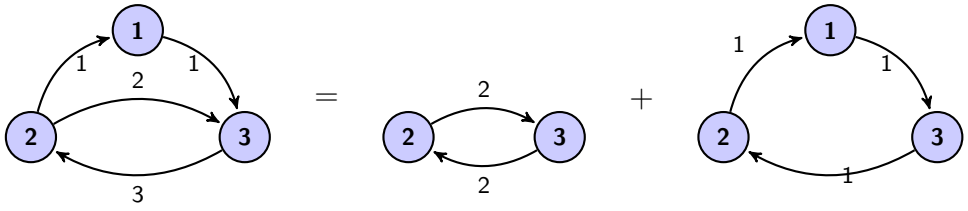


Fig. 5: Decomposition of 1(b) transaction graph.

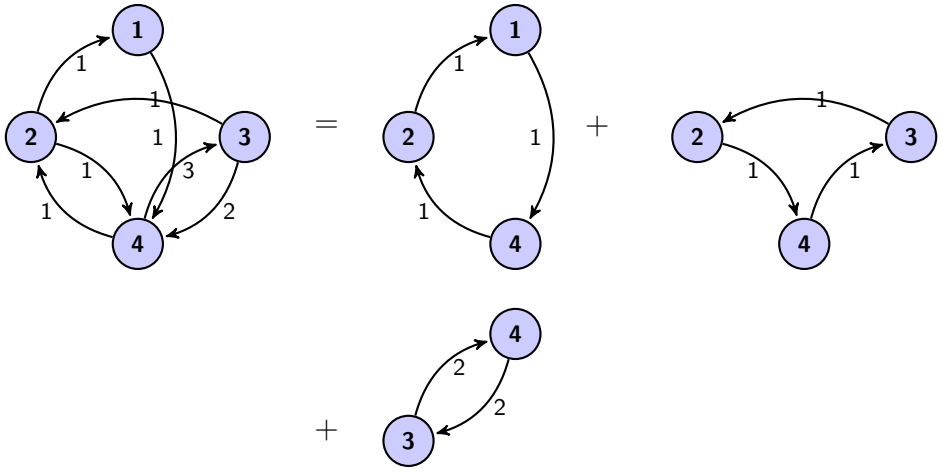


Fig. 6: Decomposition of 1(c) transaction graph.

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 failure of any given cycle's mix does not pose an exposure risk to any players with respect to the other mixes.

References

1. DarkWallet: CoinMixing. `https://wiki.unsystem.net/en/index.php/DarkWallet/CoinMixing` (2014)
2. Lynch, N.: Distributed Algorithms. Morgan Kaufmann (1996)
3. Maxwell, G.: CoinJoin: Bitcoin privacy for the real world. `https://bitcointalk.org/index.php?topic=279249.0` (2013)
4. Maxwell, G.: CoinSwap: Transaction graph disjoint trustless trading. `https://bitcointalk.org/index.php?topic=321228.0` (2013)
5. Ruffing, T., Moreno-Sanchez, P., Kate, A.: CoinShuffle: Practical decentralized coin mixing for Bitcoin. In: ESORICS'14. Lecture Notes in Computer Science, vol. 8713, pp. 345–364. Springer (2014)
6. Sun, L., Wang, M.: An algorithm for a decomposition of weighted digraphs — with applications to life cycle analysis in ecology. Journal of Mathematical Biology 54, 199–226 (2007)