P2P Mixing and Unlinkable P2P Transactions[*]

Scalable Strong Anonymity without External Routers

Pedro Moreno-Sanchez
Purdue University
pmorenos@purdue.edu

Tim Ruffing
CISPA, Saarland University
tim.ruffing@cispa.saarland

Aniket Kate
Purdue University
aniket@purdue.edu

ABSTRACT

Starting with Dining Cryptographers networks (DC-net), several peer-to-peer (P2P) anonymous communication protocols have been proposed. Despite their strong anonymity guarantees none of those has been employed in practice so far: Most fail to simultaneously handle the crucial problems of slot collisions and malicious peers, while the remaining ones handle those with a significant increased latency (communication rounds) linear in the number of participating peers in the best case, and *quadratic* in the worst case. In this work, we conceptualize these P2P anonymous communication protocols as *P2P mixing*, and present a novel P2P mixing protocol, Fast-DC, that only requires constant (i.e., four) communication rounds in the best case, and $4 + 2f$ rounds in the worst case of f malicious peers. As every individual malicious peer can halt a protocol run by omitting its messages, with its worst-case linear-round complexity, we find Fast-DC to be an optimal P2P mixing solution.

We find Fast-DC to be an ideal privacy-enhancing primitive for Bitcoin and other emerging P2P payments alternatives. Public verifiability of their pseudonymous transactions through publicly available ledgers (or blockchains) makes these systems highly vulnerable to a variety of linkability and deanonymization attacks. Fast-DC can allow pseudonymous users to make their transactions unlinkable to each other in a manner *fully compatible* with the existing systems. As representative examples, we extend Fast-DC to create unlinkable Bitcoin and Ripple path-based settlement transactions. We also demonstrate practicality of Fast-DC with a proof-of-concept implementation, and find it to require less than 16 seconds even with 50 users in a realistic environment.

Keywords

coin mixing, crypto-currencies, credit networks, anonymity, dining cryptographers networks

1. INTRODUCTION

Chaum [16] introduced the concept of anonymous digital communication in the form of mixing networks (or *mixnet*) in 1981. In the mixnet protocol, a batch of encrypted messages from users are decrypted, randomly permuted, and relayed by a sequence of trusted routers to avoid individual messages from getting traced through the network. Motivated from this seminal work, Goldschlag, Reed and Syverson [25] propose *onion routing* to achieve low-latency anonymous communication over the Internet, which culminated into the original onion routing project [26] and many other low-latency anonymous communication networks such as AN.ON [7] and I2P [32]. Among these the second generation onion routing network Tor [19] has turned out to be a definite success: by now, it protects privacy of millions of users using more than seven thousand dedicated proxies all over the world.

Almost simultaneously, we are observing an unprecedented growth of peer-to-peer (P2P) cryptocurrency systems such as Bitcoin [46] as well as transaction settlement networks such as Ripple [51] and Stellar [4]. These payment systems employ pseudonymity in an attempt to ensure that a user's accounts are not associated with her real-world identity. Along with their decentralized nature, this promise of privacy has been pivotal to success of these payment systems so far: Bitcoin market capitalization has been in billions dollars for the last few years consistently, and several banks are adopting Ripple as their transaction backbone [31, 37, 38, 52, 53, 55].

However, this perception of privacy has however turned out to be incorrect. By applying suitable heuristics to the Bitcoin public ledger (or the blockchain), Bitcoin transactions sent and received by a particular user are observed to be easily linkable [6, 9, 35, 41, 42, 47, 56]. A recent work [45] has demonstrated similar linkability issues with IOweYou (IOU) settlement networks such as Ripple [51], which also rely on pseudonymous accounts and a public transaction ledger. As a result, given the inherent public nature of payments, these emerging payment system are now considered to provide lesser privacy than traditional banking.

One line of research proposes entirely new privacy-preserving payment protocols [10, 34, 43, 44] foregoing compatibility with the existing systems; however, their regulatory feasibility and real-world success is yet to be determined.

A second line of research aims at improving privacy within existing systems [1, 9, 11, 12, 30, 40, 54, 59, 62] to ensure backwards compatibility. While the deployed anonymous communication networks such as Tor has been employed in some of these solutions [39] they are at odds with the emerging P2P payment systems: Most OR and mixnet systems including Tor inherently rely on dedicated, altruist parties (or routers) for privacy, integrity or liveness, whereas the core philosophy of payments systems is to aim at a P2P network requiring no external trusted or untrusted node. Some solutions such as CoinShuffle [54], inspired from a P2P anonymous communication system Dissent [18, 58], instead propose tailored P2P protocols that do not require any trusted or untrusted party and are becoming popular in crypto-currencies.

[*]This is a draft (revision 2016-06-03); the most recent revision is available at https://crypsys.mmci.uni-saarland.de/projects/FastDC/.

Nevertheless, with communication rounds linear in the number of participating peers in the best case and quadratic in the worst case, these current pure P2P solutions [18,54,58] are not scalable as the number of participating peers grow. Moreover, none of the existing solutions are compatible with payment settlement networks such as Ripple.

Our Contributions. Motivated from the privacy requirement of real-world P2P payment systems, as our first contribution, we introduce the concept of peer-to-peer (P2P) mixing as a natural generalization of the dining cryptographers network (DC-net) [15] protocol (Section 2). A P2P mixing protocol allows a set of peers to publish their messages anonymously without requiring any external (trusted or untrusted) party such that the anonymity set equal to the number of honest peers.

Although some extensions of the DC-net protocol [24,29,36] as well as some anonymous group messaging systems [18, 54,58] also satisfy the P2P mixing requirements, we found those to be not efficient enough for a large-scale mixing and present the new P2P mixing protocol Fast-DC (Section 3). Fast-DC builds on the original DC-net protocol, and handles collisions and malicious peers using an interesting privacy-preserving redundant messaging step. When every peer behaves honestly, Fast-DC requires only a *constant* (i.e., four) number of rounds, and it only increases to $4+2f$ rounds in the worst case, even in the presence of f malicious peers. Both of these communication round complexities are at least a linear factor better than the existing approaches [18,24,29,36,54].

We provide a proof-of-concept implementation of the Fast-DC protocol, and evaluate it in a realistic Emulab [61] network scenario. Our results show that even 50 peers can anonymously broadcast their messages in less than 16 seconds, demonstrating that the techniques employed in Fast-DC can be used to improve efficiency of mixing systems.

As our second contribution, we instantiate Fast-DC with the two most prominent P2P payment systems Bitcoin and Ripple. In particular, for Bitcoin, building on the CoinJoin paradigm [40] and Fast-DC, we present CoinShuffle++, a practical decentralized mixing protocol for the Bitcoin users (Section 4). CoinShuffle++ not only is considerably simpler and thus easier to implement than CoinShuffle [54] but also inherits the efficiency of Fast-DC and thus outperforms its CoinShuffle significantly. We also find CoinShuffle++ to be compatible with other currently deployed privacy-preserving P2P currency systems [3] and provide detailed description to ease deployment in crypto-currency clients.

For the Ripple IOU settlement network, we propose CreditMix, the first practical decentralized P2P mixing protocol for the path-based transactions (Section 5). CreditMix uses Fast-DC and adds only a constant number of rounds for the peers to successfully carry out their desired settlement transactions anonymously over the intersecting paths. CreditMix is fully compatible with Ripple as demonstrated by our proof-of-concept implementation, where we have successfully carried out a realistic mixing transaction following the CreditMix protocol over the real Ripple network.

Finally, with its well-defined generic communication interface, Fast-DC is applicable to P2P communication systems in general, and we propose it also for non-payment applications such as Sybil-resistant pseudonymization [22].

Organization. The paper is organized as follows. We start by defining the problem of P2P mixing (Section 2). We then detail the Fast-DC protocol and evaluate its performance (Section 3). We describe the use of Fast-DC to achieve anonymous transactions in Bitcoin (Section 4) as well as in Ripple (Section 5), and we conclude our work (Section 7).

2. PROBLEM AND SOLUTION OVERVIEW

A P2P mixing protocol [18,54,62] allows n peers, each peer p having a message m_p, to send their set of messages anonymously (i.e., breaking the linkability between m_p and p) without relying on any third-party router or proxy.

Security Requirements. The key requirement of a P2P mixing protocol is *sender anonymity*, i.e, the protocol should hide the relations between individual messages m_p and their owners p even from colluding malicious peers in the protocol, and from the network attacker. The anonymity set of an individual honest peer should be the set of all honest peers.

The second requirement of a P2P mixing protocol is *termination*, i.e., the protocol should terminate even in the presence of malicious peers who send wrong protocol messages (active disruption) or refuse to send any message (crash failure). While the protocol cannot force malicious peers to behave honestly, it should guarantee that eventually all honest peers deliver a set of final messages that contains at least the messages of honest peers.

Confirmation. A core feature of a P2P mixing protocol is that it provides each peer with an explicit *confirmation* on the final message M obtained from all peers, e.g., in form of a set of signatures on M, one by each peer. The form of the confirmation depends on the context and application and is left to be defined by the user. Again, while the protocol cannot force malicious peers to confirm M, those malicious peers should be excluded and the protocol should provide confirmation that all non-excluded peers agree on the final set of messages.

If necessary, the protocol is allowed to try again by obtaining new messages from peers while discarding the old messages. In this case, sender anonymity is not guaranteed for the discarded messages, which seems to put privacy at risk. However, it turns out that in a variety of applications, this is not a problem at all and a P2P mixing protocol is a very useful primitive. Moreover, our permissive way of defining a P2P mixing protocol allows for a very efficient construction.

2.1 Communication Setting and Assumptions

Communication Setting. We assume that peers are connected via a bulletin board, e.g., a server receiving messages from each peer and broadcasting them to all other peers.

Further, we assume the bounded synchronous communication setting, where time is divided in fixed communication rounds such that all messages broadcast by a peer in a round are available to the peers by the end of the same round, and absence of a message on the bulletin board indicates that the peer in question failed to send a message during the round.

Such a bulletin board can seamlessly be deployed in practice, and in fact even already deployed Internet Relay Chat (IRC) servers suffice. To be able to tolerate faulty or disruptive bulletin boards, it is possible to add redundancy by runing the same instance of the P2P mixing protocol on multiple bulletin boards. Notice that the bulletin board can alternatively be implemented by a reliable broadcast protocol [57] if one accepts the performance penalty.

We assume that all peers willing to participate in a P2P mixing protocol have generated (possibly temporary) key pairs of a digital signature scheme, and that they know each other's verification keys at the beginning of a protocol run.

Threat Model. For sender anonymity, we assume that the attacker controls the network (the bulletin board) and $f < n-1$ peers. The anonymity set of each honest peer will be the set of honest peers, and the bound $f < n-1$ implies that at least two honest peers are present to get any meaningful anonymity guarantee.

For termination (or liveness), we additionally assume that the attacker does not control the bulletin board as termination is clearly impossible in presence of an attacker who can block communication.

To find other peers willing to mix messages, a suitable bootstrapping mechanism can be used. Note that a malicious bootstrapping mechanism might hinder sender anonymity by excluding honest peers and thereby forcing a victim peer to run the P2P mixing protocol with no or only a few honest peers. While this is a realistic threat against any AC protocol in general, we consider protection against a malicious bootstrapping mechanism orthogonal to our work.

Application Interface. To deploy a P2P mixing protocol in various applications, our generic definition leaves it up to the user to specify how new input messages are obtained and how the confirmation on the output is performed.

A protocol instance consists of one or several *runs*, each started by calling the user-defined algorithm MESSAGEGEN() to obtain an input message to be mixed. When the protocol has obtained a candidate result, i.e., a candiate *output set* M of messages, it *delivers* M by calling the user-defined subprotocol CONFIRM(i, P, M), whose task is to obtain confirmation for M from the *final peer set* P of all unexcluded peers. (And i is an identifier of the run.) Possible confirmations range from a signature on M, to a complex task requiring interaction among the peers.

If confirmation can be obtained from everybody, then the run and the P2P mixing protocol *terminates successfully.* Otherwise, CONFIRM(i, P, M) by convention fails and reports peers deviating from the confirmation steps back to the P2P mixing protocol. In this case, a new run is possible, and the protocol excludes the malicious peers from this further run.[1]

Remark. A P2P mixing protocol might omit our application interface and instead delegate on an application layer to handle confirmation, and in case of failure, start a second instance of P2P mixing protocol after excluding peers refusing confirmation. This approach, however, is inherently sequential: the second instance can be started only after the confirmation of the first instance, since the set of excluded peers is not determined earlier. Instead, our treatment enables concurrent runs for improved efficiency (see Section 3): it is possible to switch to an already started second run when it turns out that the first run will fail to confirm.

2.2 Security Goals

We are now ready to state the security properties of a P2P mixing protocol. Recall that the attacker controls up to $f < n-1$ peers.

Sender Anonymity. If the protocol succeeds for honest peer p in a run with message m_p and final peer set P, and $p' \in P$ is another honest peer, then the attacker cannot distinguish whether message m_p belongs to p or to p'.

Termination. If the bulletion board is reliable, the protocol eventually terminates successfully for honest peer.

2.3 Overview on our Solution

Our core tool to design an efficient P2P mixing protocol is a Dining Cryptographers Network (DC-net) [15].

We describe an example of a DC-net involving two users. Suppose that two peers p_1 and p_2 share a key k and that one of the peers (e.g., p_1) wishes to anonymously publish a message m such that $|m| = |k|$. For that, p_1 publishes $M_1 := m \oplus k$ and p_2 publishes $M_2 := k$. An observer can compute $M_1 \oplus M_2$, effectively recovering m. The origin of this message is hidden: without knowing the secret k, the observer cannot determine which peer published m. We refer the reader to [27] for details on how to extend this basic protocol to multiple users.

DC-net in the Real World. Besides the need for pairwise symmetric keys, which can be overcome by a key exchange mechanism, there are three key challenges to deploy a DC-net in practice with an arbitrary number of peers, namely first, handling collisions when more than one peer try to publish her message; second, ensuring agreement on the outcome of the protocol among honest peers even in the presence of malicious peers; and third, handling of disruptive peers.

Handling Collisions. Each peer $p \in P$ in the mixing seeks to anonymously publish her own message m_p. Naively, they could run $|P|$ instances of a DC-net, where each peer randomly selects one instance (or slot) to publish her message. However, even if all peers are honest, two peers can choose the same slot with high probability, and their messages are unrecoverable [27].

One proposed solution consists on performing an anonymous slot reservation mechanism so that peers agree in advance on an ordering for publishing [24,29,36]. However, this mechanism adds communication rounds among the peers and these reservation schemes must handle collisions on themselves. Alternatively, it is possible to set many more slots so that probability of collision decreases [17]. However, this becomes inefficient quickly, and two honest peers could still collide with some probability.

Instead, we handle collisions by redundancy [13,20]. Assume that messages to be mixed are encoded as elements of a finite field $\mathbb{F}$ with characteristic greater than the number of peers n. Given n slots, each peer i, with message m_i, publishes m_i^j in the j-th slot. This results in having a collision from all peers for each of the slots. Using addition in $\mathbb{F}$ instead of XOR to create DC-net messages, the j-th collision results on the i-th power sum $S_i = \sum_i m_i^j$.

Now, we require a mechanism to extract the messages m_i from the power sums S_i. Let $f(x) = x^n + a_{n-1}x^{n-1} + \ldots + a_1x + a_0$ be the polynomial with roots $m_1, \ldots, m_n$. Newton's identities [21,28] define linear relations between a_i and S_i. Since every peer knows $\{S_i\}$, they can easily calculate the a_i, reconstruct the polynomial $f(x)$, and then factorize it. The n roots are the messages m_i from each peer. Intuitively, symmetry of the power sums ensures sender anonymity.

Handling Disruptions. Recovering the messages only works when all peers honestly follow the protocol. However, a peer could disrupt the DC-net by simply using inconsistent DC-net messages. In this case, the final set M does not contain messages from honest peers, and the honest peers

[1]Instead of dealing with multiple runs explicitly, we could delegate to responsibility to start a new run in case of failure to the application layer. However, it turns out that our treatment will allow for more efficient constructions that make use of concurrent runs.

can deny confirmation.

To detect and exclude the misbehaving peer every peer is required to reveal the secret key used in the initial key exchange. Then every peer can replay the steps done by every other peer and eventually detect and expel the misbehaving peer from a new run.

Note that the revelation of the secret keys clearly breaks anonymity for the current run of the protocol. However, the failed run will be discarded and a new run with fresh cryptographic keys and new messages is will be started without the misbehaving peer. This is in line with our definition of sender anonymity, which does not impose a requirement on failed runs; it will become clear why this is not a problem.

Termination. To ensure that the protocol terminates for all honest peers, we must ensure that all honest peers agree on the current state and stage of the protocol, even in the presence of malicious peers.

It turns out that the crucial point in our approach is after determining the output set M. At this stage, every honest peer checks whether its the provided input message is in M. Depending on the outcome of this check, the peer either starts the confirmation subprotocol to confirm a good M, or reveals the secret key used in the key exchange to determine who is responsible for an incorrect M. For termination, it is crucial that all honest peers take the same decision at this stage of the protocol.

To overcome this challenge, every peer must provide a non-malleable commitment (e.g., using a hash function) to its DC-net vector before it sees the vectors of other peers. In this manner, malicious peers are forced to create their DC-net vectors independently of the DC-net vector (and the unpredictable input messages) of honest peers. Thus, the redundant encoding of messages ensures that a malicious peer is not able to create a malformed DC-net vector that results in a distortion of only a subset of the honest peers. Intuitively, to distort some messages but keep some other message m of a honest peer intact, the malicious peer must influence the all power sums consistently. This would requires a DC-net vector that depends on m. This ensures that all honest peers agree on whether M is correct or not, and take the same control flow decision.

Another important guarantee provided by Fast-DC is that if a protocol run fails, the honest peers agree on the set of malicious peers to be excluded. Although this is critical for termination, this aspect has not been properly formalized and addressed on previous P2P mixing protocols [18, 58] supposed to ensure termination.

3. THE FAST-DC PROTOCOL

In this section we present Fast-DC, a very efficient P2P mixing protocol, which terminates in only $4+2f$ rounds in the presence of f malicious peers.

3.1 Building Blocks and Conventions

Digital Signatures. We require a digital signature scheme (KeyGen, Sign, Verify) unforgeable under chosen-message attacks (UF-CMA). The algorithm KeyGen returns a private signing key sk and the corresponding public verification key vk. On input message m, $\mathsf{Sign}(sk, m)$ returns σ, a signature on message m using signing key sk. The verification algorithm $\mathsf{Verify}(pk, \sigma, m)$ outputs $true$ iff σ is a valid signature for m under the verification key vk.

Non-interactive Key Exchange. We require a non-interactive key exchange (NIKE) mechanism (NIKE.KeyGen, NIKE.SharedKey) secure in the CKS model [14,23]. The algorithm $\mathsf{NIKE.KeyGen}(id)$ outputs a public key npk and a secret key nsk for given a party identifier id. $\mathsf{NIKE.SharedKey}(id_1, id_2, nsk_1, npk_2, sid)$ outputs a shared key for the two parties and session identifier sid. NIKE.SharedKey must fulfill the standard correctness requirement that for all session identifiers sid, all parties id_1, id_2, and all corresponding key pairs (npk_1, nsk_1) and (npk_2, nsk_2), it holds that $\mathsf{NIKE.SharedKey}(id_1, id_2, nsk_1, npk_2, sid) = \mathsf{NIKE.SharedKey}(id_2, id_1, nsk_2, npk_1, sid)$. Additionally, we require an algorithm $\mathsf{NIKE.ValidatePK}(npk)$, which outputs $true$ iff npk is a public key in the output space of NIKE.KeyGen, and we require an algorithm $\mathsf{NIKE.ValidateKeys}(npk, nsk)$ which outputs $true$ iff nsk is a secret key for the public key npk.

Static Diffie-Hellman key exchange satifies these requirements [14], given a suitable key derivation algorithm such as $\mathsf{NIKE.SharedKey}(id_1, id_2, x, g^y) := \mathsf{K}((g^{xy}, \{id_1, id_2\}, sid))$ for a hash function K modeled as a random oracle.

Hash Functions. We require two hash functions H and G both modeled as a random oracle.

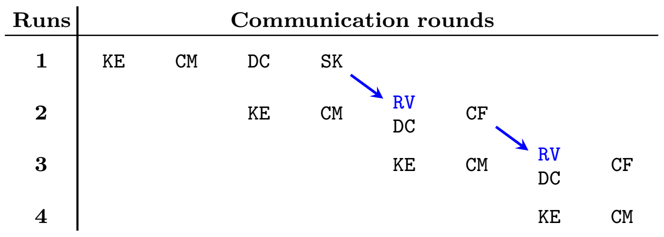
Pseudocode Conventions. We use arrays written as $\textsc{arr}[i]$, where i is the index. We denote the full array (all its elements) as $\textsc{arr}[\,]$.

Message x is broadcast using "**broadcast** x". The command "**receive** $\mathrm{X}[p]$ **from all** $p \in P$ **where** $X(\mathrm{X}[p])$**missing** $C(P_{\mathit{off}})$" attempts to receive a message from all peers $p \in P$. The first message $\mathrm{X}[p]$ from peer p that fulfills predicate $X(\mathrm{X}[p])$ is accepted and stored as $\mathrm{X}[p]$; all further messages from p are ignored. When a timeout is reached, the command C is executed, which has access to a set $P_{\mathit{off}} \subseteq P$ of peers that did not send a (valid) message.

Regarding concurrency, a thread that runs a procedure $\mathrm{P}(args)$ is started using "$t :=$ **fork** $\mathrm{P}(args)$", where t is a handle for the thread. A thread t can either be joined using "$r :=$ **join** t", where r is its return value, or it can be aborted using "**abort** t". A thread can wait for a notification and receive a value from another thread using "**wait**". The notifying thread uses "**notify** t **of** v" to notify thread t of some value v.

3.2 Contract with the Application

Guarantees Provided to the Application. To ensure



Runs	Communication rounds							
1	KE	CM	DC	SK				
2			KE	CM	RV DC	CF		
3					KE	CM	RV DC	CF
4							KE	CM

Figure 1: A Fast-DC execution. Run 1 fails due to DC-net disruption. Run 2 fails to confirm. Run 3 finally succeeds and run 4 is then aborted. Rows represent protocol runs and columns represent communication rounds. Blue parts are for concurrency; arrows depict dependencies between rounds. KE: Key exchange; CM: Commitment; DC: DC-net; RV: Reveal pads; SK: Reveal secret key; CF: Confirmation.

that noone can refuse confirmation for a legitimate reason, e.g., an incorrect set M not containing her message, the P2P mixing protocol ensures that all honest peers deliver the same and correct message set M. As a result it is safe to assume that peers refusing to confirm are indeed malicious.

Agreement: Let p and p' be two honest peers in a protocol instance. If p calls $\textsc{Confirm}(i, P, M)$[2] in some communication round r, then p' calls $\textsc{Confirm}(i, P, M)$ with the same message set M and final peer set P in the same communication round r.

Validity: If honest peer p calls $\textsc{Confirm}(i, P, M)$ with message set M and final peer set P, then *(i)* for all honest peers p' and their messages $m_{p'}$, we have $m_{p'} \in M$, and *(ii)* we have $|M| = |P|$.

Requirements on the Application. We require three natural properties from the confirm subprotocol. The first property states that a successful call to the subprotocol indeed confirms that the peers in P agree on M. The second property states that in an unsuccessful call, no honest peer is falsely accused as a malicious peer. The third property states that in an unsuccessful call, the honest peers agree who refused confirmation maliciously.

Confirmation: Even if the bulletin board is malicious, if a call to $\textsc{Confirm}(i, P, M)$ succeeds for peer p, then all honest peers in P have called $\textsc{Confirm}(i, P, M)$.

No Honest Exclusion: If $\textsc{Confirm}(i, P, M)$ returns a set $P_{malicious}$, then $P_{malicious}$ does not contain honest peers.

Agreement after Failure: If $\textsc{Confirm}(i, P, M)$ returns a set $P_{malicous} \neq \emptyset$ for peer p, then $\textsc{Confirm}(i, P, M)$ returns the same set $P_{malicious}$ for every honest peer p'.

Encoding of Input Messages. We assume that input messages generated by $\textsc{MessageGen}()$ are encoded in a prime field $\mathbb{F}_q$, where q is larger than the number of peers in the protocol. Also, we assume w.l.o.g. that the message m returned by $\textsc{MessageGen}()$ has sufficient entropy such that it can be predicated only with negligible probability. (This can be ensured by a randomized encoding such as encoding m as $m||r$ for a sufficiently large string r.) Note that this in particular that q is at least as large as the security parameter.

3.3 Protocol Description

We describe the Fast-DC protocol in Algorithms 1 and 2. The black code is the basic part of the protocol; the blue code is necessary to handle several runs concurrently and to handle offline peers.

Single Run of the Protocol (Black Pseudocode). The protocol starts in $\textsc{Start-Fast-DC}()$, which receives as input a set of other peers P, our own identity *my*, an array $\mathrm{VK}[\,]$ of verification keys of the other peers, our own signing key *sk*, and a predetermined session identifier *sid*.

A single run of Fast-DC consists of four rounds. ($\textsc{Run}()$) The first three rounds are key exchange (`KE`), publishing of commitments to the DC-net messages (`CM`) and publishing of such DC-net messages (`DC`). The fourth round consists on accepting the result in the absence of disruptions (`CF`) or the publishing of secret keys to discover the misbehaving peer (`SK`).

The first round (`KE`) just uses the NIKE to establish pairwise symmetric keys between all peers ($\textsc{DC-Keys}()$). Then each peer can derive the DC-net pads from these symmetric keys ($\textsc{DC-Pad}()$).

In the second round (`CM`), each peer commits to her DC-net vector using hash function `H`; adding randomness is not necessary, because we assume that the input messages contained in the DC-net vector have sufficient entropy. The commitments are opened in the third round (`DC`). They are non-malleable and their purpose is to prevent a rushing attacker from letting his DC-net vector depend on messages by honest peers, which will be crucial for the agreement property. After opening the commitments, every peer has enough information to solve the DC-net and extract the list of messages by solving the power sums ($\textsc{DC-Res}()$).

Finally, every peer checks whether her input message is in the result of the DC-net. Agreement will ensure that either every peer finds her message or no honest peer finds it.

If a peer finds her message, she proceeds to the confirmation subprotocol. Otherwise, she outputs her secret key. In this case, every other peer publishes her secret key as well, and all peers can replay each other protocol messages for the current run. This will expose the misbehaving peer and honest peers will exclude him from the next run.

Concurrent Runs of the Protocol (Blue Pseudocode). To reduce the number of communication rounds to $4 + 2f$, we deploy concurrent runs as exemplified in Figure 1. We need to address two main challenges. First, when a peer disrupts the DC-net phase of run i, it must be possible to "patch" the already started run $i + 1$ to discard messages from misbehaving peers in run i. For that, run i must reach the last phase (`SK` or `CF`) before run $i + 1$ reaches `DC` phase.

Until run $i+1$ sends the `DC` message, it can can be patched as follows. In the `DC` phase of run $i + 1$, honest peers broadcast not only their DC-net messages, but also in parallel they reveal (`RV`) the symmetric keys shared in run $i + 1$ with malicious peers detected in run i. In this manner, DC-net messages can be partially unpadded, effectively excluding misbehaving peers from run $i + 1$. We note that, a peer could reveal wrong symmetric keys in the `RV` step. This, however, leads to wrong output of the DC-net, which is then handled by revealing secret keys in round $i + 1$. Moreover, publishing partial symmetric keys does not compromise sender anonymity since messages remain partially padded with symmetric keys shared between the honest peers.

Handling Offline Peers (Blue Pseudocode). So far we have only discussed how to ensure termination against actively disruption peers who send wrong messages. However, a malicious peer can also just send no message at all. This case is easy too handle in our protocol. If a peer p has not provided a (valid) broadcast message to the bulletin board (in time), all honest peers will agree on that fact, and exclude the unresponsive peer. In particular, it is easy to see that all criteria specifying whether a message is valid will evaluate the same for all honest peers (if the bulletin board is reliable, which we assume for termination).

Observe that for missing messages from the first two broadcasts (`KE` and `DC`), the current run can be continued. Peers not sending `KE` are just ignored in the rest of the run; peers not sending `CM` are handled by revealing symmetric keys exactly as done with concurrent runs. Observe that this crucially

[2] $\textsc{Confirm}(\ldots)$ will actually take more arguments but they are not relevant for this subsection.

ensures that even in the presence of passively disrupting peers, only $4 + 2f$ communications rounds are necessary.

3.4 Security Analysis

In this section, we discuss why Fast-DC achieves all required properties.

Sender Anonymity. Consider a protocol execution in which a honest peer p succeeds with message m_p and final peer set P, and let $p' \in P$ be another honest peer. We have to argue that the attacker cannot distinguish whether m_p belongs to p or p'.

Since both p and p' choose fresh messages m_p,$m_{p'}$, and fresh NIKE key pairs in each run, it suffices to consider only the successful run i. Since p succeeds in run i, the call to $\textsc{Confirm}(i, P, M)$ has succeeded. By the confirmation property of $\textsc{Confirm}(\ldots)$, p' has started $\textsc{Confirm}(i, P, M)$ in the same communication round as p. By construction of the protocol, this implies two properties about p': *(i)* p' will not reveal her secret key in round SK. *(ii)* peer p' assumes that p is not excluded in run i, and thus has not revealed the symmetric key shared with p in round RV. (Part *(ii)* is only relevant in the concurrent variant of the protocol.)

As the key exchange scheme is secure in the CKS model and the public keys are authenticated using signatures, the attacker cannot distinguish the random DC-nets derived from the symmetric key between p and p' from random pads.

Thus, after opening the commitments on the pads, p has formed a proper DC-net with at least p'. The security guarantee of original Chaum's DC-nets [15] implies that the attacker cannot distinguish m_p from $m_{p'}$ before the call to $\textsc{Confirm}(i, P, M)$. Now, observe that the execution of subprotocol $\textsc{Confirm}(i, P, M)$ does not help in distinguishing, since all honest peers call it with the same arguments, which we have already argued.

Termination. We show why the protocol terminates for every honest peer. On the way, we will further show agreement, and validity. Recall that we assume the bulletin board to be reliable for termination, so every peer receives the same broadcast messages. Under this assumption and the assumption that the signature scheme is unforgeable, a code inspection shows that after receiving the DC message, the entire state of a protocol run i is the same for every honest peer, except for the signing keys, the own identity my, and the message m generated by $\textsc{MessageGen}()$. From these three items, only m influences the further state and control flow, and it does so only in the check $m \in M$.

Consequently, when considering run i, it suffices to show that the condition $m \in M$ is either true for all honest peers or is false for all honest peers. (Note that also M is entirely determined by broadcast messages and thus the same for all honest peers.)

Let p and p' be two honest peers with their input messages m_p and $m_{p'}$ in run i, and assume for contradiction that the condition is true for p but not for p', i.e., $m_p \in M$ but $m_{p'} \notin M$. This implies that at least one malicious peer a has committed to an ill-formed DC-net vector in run i, i.e., a vector which is not of the form $(m_a, m_a^2, \ldots, m_a^n)$ with $n \geq 2$. Since $m_p \in M$, this ill-formed vector left the message m_p intact. A straight-forward argument involving the power sums shows that it is infeasible to create such an ill-formed vector without information about m_p.

As the message $\mathsf{H}(\mathtt{dc}, \mathrm{DC}[\,])$ implements a hiding, binding

Algorithm 1 Fast-DC: Main Protocol Run

```
procedure $\textsc{Start-Fast-DC}(P, my, \mathrm{VK}[\,], sk, sid)$
    $sid' := (sid, P, \mathrm{VK}[\,])$
    if $my \in P$ then
        fail "cannot run protocol with myself"
    $\textsc{Run}(P, my, \mathrm{VK}[\,], sk, sid', 0)$

procedure $\textsc{Run}(P, my, \mathrm{VK}[\,], sk, sid, run)$
    if $P = \emptyset$ then
        fail "no honest peers "

    $\triangleright$ Exchange pairwise keys
    $(\mathrm{NPK}[my], \mathrm{NSK}[my]) := \mathsf{NIKE.KeyGen}(my)$
    $sidH := \mathsf{H}((\mathtt{sid}, sid, P \cup \{my\}, \mathrm{NPK}[\,], run))$
    broadcast $(\mathtt{KE}, \mathrm{NPK}[my], \mathsf{Sign}(sk, (\mathrm{NPK}[my], sidH)))$
    receive $(\mathtt{KE}, \mathrm{NPK}[p], \sigma[p])$ from all $p \in P$
        where $\mathsf{NIKE.ValidatePK}(\mathrm{NPK}[p])$
                $\wedge\ \mathsf{Verify}(\mathrm{VK}[p], \sigma[p], (\mathrm{NPK}[p], sidH))$
    missing $P_{off}$ do
        $P := P \setminus P_{off}$                    $\triangleright$ Exclude offline peers

    $\triangleright$ Create fresh message to mix and prepare DC-net
    $m := \textsc{MessageGen}()$
    $\mathrm{K}[] := \textsc{DC-Keys}(P, \mathrm{NPK}[\,], my, \mathrm{NSK}[my], sidH)$
    $\mathrm{DC}[my][\,] := \textsc{DC-Vector}(P, \mathrm{K}[\,], m)$

    $\triangleright$ Set of malicious (and offline) peers for later exclusion
    $P_{excl} = \emptyset$

    $\triangleright$ Commit to DC-net vector
    $\textsc{Com}[my] := \mathsf{H}((\mathtt{dc}, \mathrm{DC}[my]))$
    broadcast $(\mathtt{CM}, \textsc{Com}[my], \mathsf{Sign}(sk, (\textsc{Com}[my], sidH)))$
    receive $(\mathtt{CM}, \textsc{Com}[p], \sigma[p])$ from all $p \in P$
        where $\mathsf{Verify}(\mathrm{VK}[p], \sigma[p], (\textsc{Com}[p], sidH))$
    missing $P_{off}$ do              $\triangleright$ Store offline peers for exclusion
        $P_{excl} := P_{excl} \cup P_{off}$
    if $run > 0$ then
        $\triangleright$ Wait for prev. run to notify us of malicious peers
        $P_{exclPrev}$ := wait
        $P_{excl} := P_{excl} \cup P_{exclPrev}$

    $\triangleright$ Collect shared keys with excluded peers
    for all $p \in P_{excl}$ do
        $\mathrm{K}_{excl}[my][p] := \mathrm{K}[p]$
    $\triangleright$ Start next run (in case this one fails)
    $P := P \setminus P_{excl}$
    $next :=$ fork $\textsc{Run}(P, my, \mathrm{VK}[], sk, sid', run + 1)$

    $\triangleright$ Open commitments  and keys with excluded peers
    broadcast $(\mathtt{DC}, \mathrm{DC}[my][\,], \mathrm{K}_{excl}[my][\,], \mathsf{Sign}(sk, \mathrm{K}_{excl}[my][\,]))$
    receive $(\mathtt{DC}, \mathrm{DC}[p][\,], \mathrm{K}_{excl}[p][\,], \sigma[p])$ from all $p \in P$
        where $\mathsf{H}((\mathtt{dc}, \mathrm{DC}[p][\,])) = \textsc{Com}[p]$
                $\wedge\ \{p' : \mathrm{K}_{excl}[p][p'] \neq \bot\} = P_{excl}[p]$
                $\wedge\ \mathsf{Verify}(\mathrm{VK}[p], \mathrm{K}_{excl}[p][\,], \sigma[p])$
    missing $P_{off}$ do                 $\triangleright$ Abort and rely on next run
        return $\textsc{Result-Of-Next-Run}(P_{off}, next)$

    $\triangleright$ Check if our output is contained in the result
    $M := \textsc{DC-Res}(P \cup \{my\}, \mathrm{DC}[\,][\,], P_{excl}, \mathrm{K}_{excl}[\,][\,])$
    if $m \in M$ then
        $P_{malicious} := \textsc{Confirm}(i, P, M, my, \mathrm{VK}[\,], sk, sid)$
        if $P_{malicious} = \emptyset$ then                          $\triangleright$ Success?
            abort $next$
            return $m$
    else
        broadcast $(\mathtt{SK}, \mathrm{NSK}[my])$              $\triangleright$ Reveal secret key
        receive $(\mathtt{SK}, \mathrm{NSK}[p])$ from all $p \in P$
            where $\mathsf{NIKE.ValidateKeys}(\mathrm{NPK}[p], \mathrm{NSK}[p])$
        missing $P_{off}$ do             $\triangleright$ Abort and rely on next run
            return $\textsc{Result-Of-Next-Run}(P_{off}, next)$
        $\triangleright$ Determine malicious peers using the secret keys
        $P_{mal} := \textsc{Blame}(\mathrm{NPK}[\,], \mathrm{NSK}[\,], \mathrm{DC}[\,][\,], sidH, P_{excl}, \mathrm{K}_{excl}[\,][\,])$

    return $\textsc{Result-Of-Next-Run}(P_{mal}, next)$
```

Algorithm 2 Fast-DC: Sub-procedures

```
procedure DC-VECTOR($P, \mathrm{K}[\,], m$)
    ▷ DC-net
    for $s := 1, \dots, |P| + 1$ do
        $\mathrm{DCMY}[s] := (m)^s + \text{DC-PAD}(P, \mathrm{K}[\,], s)$
    return $\mathrm{DCMY}[\,]$

procedure DC-KEYS($P, \mathrm{NPK}[\,], my, nsk, sidH$)
    for all $p \in P$ do
        $\mathrm{K}[p] := \mathsf{NIKE.SharedKey}(my, p, nsk, \mathrm{NPK}[p], sidH)$
    return $\mathrm{K}[\,]$

procedure DC-PAD($P, \mathrm{K}[\,], s$)
    return $\sum_{p \in P} \mathrm{sgn}(my - p) \cdot \mathsf{G}((\mathrm{K}[p], s))$        ▷ in $\mathbb{F}$

procedure DC-RES($P_{all}, \mathrm{DC}[\,][\,], P_{excl}, \mathrm{K}_{excl}[\,][\,]$)
    for $s := 1, \dots, |P| + 1$ do
        ▷ Pads cancel out for honest peers
        $\mathrm{S}[s] := \sum_{p \in P_{all}} \mathrm{DC}[p][s]$
        ▷ Also remove pads for excluded peers
        $\mathrm{S}[s] := \mathrm{S}[s] - \sum_{p \in P_{all}} \text{DC-PAD}(P_{excl}, \mathrm{K}_{excl}[p], s)$
    $\mathrm{M}[\,] := \mathsf{Solve}(\forall s \in \{1, \dots, |P|+1\}.\ \mathrm{S}[s] = \sum_{i=1}^{|P|+1} \mathrm{M}[i]^s)$
    return $\mathsf{Set}(\mathrm{M}[\,])$        ▷ Convert $\mathrm{M}[\,]$ to an (unordered) set

procedure BLAME($\mathrm{NPK}[\,], \mathrm{NSK}[\,], \mathrm{DC}[\,][\,], sidH, P_{excl}, \mathrm{K}_{excl}[\,][\,]$)
    $P_{mal} := \emptyset$
    for all $p \in P$ do
        $\mathrm{K}'[\,] := \text{DC-KEYS}(P, \mathrm{NPK}[\,], p, \mathrm{NSK}[p], sidH)$
        ▷ Purported $m$ of $p$
        $m' := \mathrm{DC}[p][1] - \text{DC-PADS}(\mathrm{K}[\,], 1)$
        ▷ Replay DC-net message of $p$
        $\mathrm{DC}'[\,] := \text{DC-VECTOR}(\mathrm{K}'[\,], m')$
        if $\mathrm{DC}'[\,] \neq \mathrm{DC}[p][\,]$ then
            $P_{mal} := P_{mal} \cup \{p\}$  ▷ Exclude inconsistent $p$
        ▷ Check if $p$ published correct symmetric keys
        for all $p_{excl} \in P_{excl}$ do
            if $\mathrm{K}_{excl}[p][p_{excl}] \neq \mathrm{K}'[p_{excl}]$ then
                $P_{mal} := P_{mal} \cup \{p\}$
    return $P_{mal}$

procedure RESULT-OF-NEXT-RUN($P_{exclNext}, next$)
    ▷ Hand over to next run and notify of peers to exclude
    notify $next$ of $P_{exclNext}$
    ▷ Return result of next run
    $result$ := join $next$
    return $result$
```

and non-malleable commitment on $\mathrm{DC}[\,]$ (recall that adding randomness is not necessary because there is sufficient entropy in $\mathrm{DC}[\,]$), it is even for a rushing malicious peer a infeasible to have committed to an ill-formed vector that leaves m_p intact. This is a contradiction, and thus the condition $m \in M$ evaluates equivalently for all honest peers. As this is exactly the condition that determines whether CONFIRM(...) is called, this shows validity for run i.

Consequently, if all honest peers started run i in the same communication round, they also call CONFIRM(...) in the same communication round, and they do so with the same arguments i, P, M. This shows agreement (just for run i).

Furthermore, all honest peers start run $i+1$ in the same communication round, and with the same set of peers P (which is the only state passed from run i to run $i+1$): If in run i, the subprotocol CONFIRM(...) is called, this holds by the assumption that the return value of CONFIRM(...) only depends on the arguments i, P, and M. If in run i, the subprotocol is not called, it holds by a straight-forward code inspection of the code responsible for determining the P for run $i+1$.

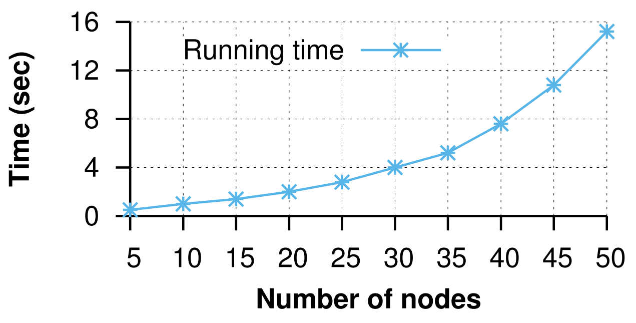


Figure 2: Running Time. Network setting: all peers in the same LAN with a bandwith of 20 Mbit/s. The bulletin board has a bandwidth of 100 Mbit/s.

As the first run starts in the first communication round for all honest peers, agreement and validity for the whole protocol follow by an inductive argument.

Finally, observe that at least one malicious peer is excluded in each run that does not succeed. In particular, if CONFIRM(...) is called in a run, then it either succeeds or it is assumed to return a non-empty set of malicious peers. If CONFIRM(...) is not called in a run, then there must be a malicious peer, and replaying all protocol messages of this run clearly identifies him. This shows termination.

3.5 Performance Analysis

Communication. Using concurrent runs, the protocol needs $(c+3)+(c+1)f$ communication rounds, where f is the number of peers actually disrupting the protocol execution, and c is the number of rounds of the confirmtion subprotocol. In the case $c = 1$ such as in our Bitcoin mixing protocol (Section 4), Fast-DC needs just $4 + 2f$ rounds. In particular it succeeds in 4 rounds if everybody behaves honestly.

The communication costs per run and per user are dominated by the broadcast of the DC-net array $\mathrm{DC}[my][\,]$ of size $n \cdot |m|$ bits, where n is the number of peers and $|m|$ is the length of a mixed message. All three other broadcasts have constant size at any given security level.

Implementation. We have developed a proof-of-concept implementation of Fast-DC based on an existing implementation of the Dissent protocol [18]. This unoptimzed implementation encompasses the complete functionality to enable testing a successful run of Fast-DC without disruptions.

The implementation is written in Python and uses OpenSSL for ECDSA signatures on the secp256k1 elliptic

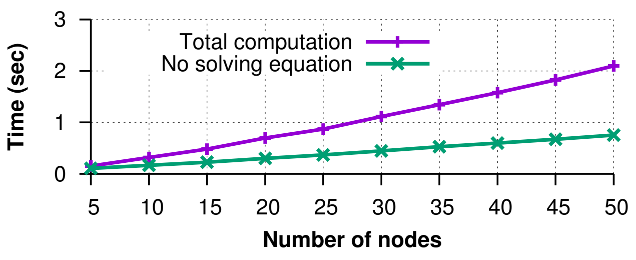


Figure 3: Computation Time. Purple line shows the total computation time. Green line shows the computation time without solving the power sums.

curve (as used in Bitcoin and Ripple) at a security level of 128 bits. We use a Python wrapper for the Pari-gp library [33] to find the roots of the power sum polynomial.

Testbed. We have tested our Fast-DC implementation in Emulab [61]. Emulab is a testbed for distributed systems that enables a controlled environment with easily configurable parameters such as network topology or bandwidth of the communication links. We simulate a network setting in which all peers (20 Mbit/s) have TCP connections to a bulletin board (100 Mbit/s). Note that the our bandwith assumption for the bulletin board is very conservative (think of an IRC server).

Results. We have run the protocol varying the number of peers, ranging from 5 to 50. Each peer has as input for the mixing a 160-bit message (e.g., a Bitcoin address). First, we have measured the overall running time of Fast-DC. As shown in Figure 2, we observe that even when 50 participants, Fast-DC takes less than 16 seconds. Second, we have studied how much time out of the total running time is spent by the peers in computation. We show our results in Figure 3. We have considered two computation times. First, the time spent by peers in creating the blinded version of their messages (i.e., green line); and second, the time required to find the polynomial's roots from the power sums (i.e., purple line). We observe that although the second time is considerably higher than the first, both are much smaller than the time spent in the communication.

These results show that even our unoptimized implementation of Fast-DC scales to a large number of peers and outperforms state-of-the-art P2P mixing solutions such as CoinShuffle [54] and Dissent [18].

4. EFFICIENT COIN MIXING IN BITCOIN

Several works [6,9,35,42,47,56] propose different heuristics to link Bitcoin payments sent or received by a particular user. Ultimately, crypto-currencies such as Bitcoin using a public Blockchain may in fact provide *less* anonymity than traditional banking. Coin mixing has emerged as a technique to overcome this problem while maintaining full compatibility with current Bitcoin protocol.

A promising solution in this direction is CoinShuffle [54], a P2P mixing protocol based on a mixnet run by the peers to ensure the unlinkability of input and output accounts in a jointly created mixing transaction (a so-called CoinJoin transaction [40]). However, a run with a decent anonymity set of $n = 50$ peers takes about three minutes to complete [54], *assuming that every peer is honest.* In the presence of f disruptive peers aiming at impeding the protocol, $O(nf)$ communications rounds are required, and most of them inevitably taking longer due to the disruptive peers delaying their messages intentionally. For say $f = 10$ disrupting peers, the protocol needs more than 30 minutes to succeed, which arguably prohibits a practical deployment of CoinShuffle. As a consequence, we lack a coin mixing protocol for crypto-currencies that is efficient enough for practical deployment.

We propose CoinShuffle++, an highly efficient coin mixing protocol resulting from the application of Fast-DC to the Bitcoin setting. In the following, we first give some background on Bitcoin. Then, we describe additional design goals when using P2P mixing protocol in payment systems, and then describe in detail our protocol. Finally, we show the experimental results from our evaluation and finally compare Fast-DC with other available proposals to overcome the anonymity problem in Bitcoin.

4.1 The Bitcoin system

Bitcoin [46] is a crypto-currency run by a P2P network. An accounts in the Bitcoin system is associated with ECDSA signing keys. The accounts are then publicly identified by a 160-bit hash of the verification key, called *address.* Every peer can create an arbitrary number of accounts by creating fresh key pairs.

A peer can spend coins stored at her account by creating and signing Bitcoin transactions. In its simplest form, a Bitcoin transaction is composed by an input account, an output account and the amount of coins to be transferred from the input to the output account. For the transaction to be successful, it must be signed with the signing key associated to the input account.

Bitcoin transactions can include multiple input and output accounts to spend coins simultaneously. In this case, the transaction must be signed with the signing keys associated to each of the input accounts.

4.2 Security Goals

Apart from the general security goals for a P2P mixing protocol (see Section 2.2), the protocol must guarantee *correct balance*, a security property of interest when the P2P mixing protocol is leveraged to mix output accounts that enable privacy preserving transactions in payment systems.

Correct Balance. If the P2P mixing protocol succeeds for peer p, then balance of p's output account is at least β (ignoring transaction fees), where β is mixing amount in the P2P mixing protocol. In no case, the total balance of p's accounts is not reduced (ignoring transaction fees).

4.3 Our Protocol

CoinShuffle++ leverages Fast-DC to perform a Bitcoin transaction where the input and output accounts for any given (honest) peer cannot be linked. In particular, CoinShuffle++ creates a fresh pair of signing-verification Bitcoin keys and returns the verification key to implement MessageGen().

Then, for the confirmation subprotocol Confirm(...), CoinShuffle++ uses CoinJoin [40, 41] to perform the actual mixing. A CoinJoin transaction allows a set of peers to mix their coins without the help of a third party. In such a transactions, peers set their current Bitcoin accounts as input and a mixed list of fresh Bitcoin accounts as output. Crucially, peers can verify whether the thereby constructed transaction transfers the correct amount of coins to their fresh output account and only if all peers agree and sign the transaction, it becomes valid. So in the case of CoinShuffle++, the explicit confirmation provided by Fast-DC is a list of valid signatures, one from each peer, on the CoinJoin transaction.

Note that Fast-DC guarantees that everybody receives the correct list of outputs account when the confirmation subprotocol is entered. So a peer refusing to sign the CoinJoin transaction can safely be considered malicious and removed.[3] This is a crucial property for an anonymous CoinJoin-based

[3]This is omitted in the pseudocode.

approach, otherwise a single malicious peer can refuse to sign the transaction and thus mount a DoS service on all other peers who cannot exclude the malicious peer if not convinced of his guilt.

We define CoinShuffle++ in Algorithm 3. There, we denote by $\mathsf{CoinJoinTx}(VK_{in}[], VK_{out}[], \beta)$ a CoinJoin transaction that transfers β bitcoins from every input to every output account (where β is a pre-arranged parameter). Moreover, we denote by $\mathsf{Submit}(tx, \sigma[\,])$ the submission of tx including all signatures to the Bitcoin network.

Security Analysis. Observe that CoinShuffle++ adheres to the requirements specified in Section 3.2. Thus, sender anonymity and termination in CoinShuffle++ are immediate. (We refer to [41] for a detailed taint-based analyses on the privacy implications of CoinJoin-based protocol providing sender anonymity.) Correct balance is enforced by the CoinJoin paradigm: by construction, a peer signs only transactions that will transfer his funds from her input address to her output address.

Performance Analysis. In our performance analysis of Fast-DC (Section 3.5), MessageGen() creates a new ECDSA key pair and Confirm(...) obtains ECDSA signatures from all peers (using their initial ECDSA key pairs) on a bitstring of 160 bits. This is almost exactly CoinShuffle++, so the performance analyses of Fast-DC carries over to CoinShuffle++.

Practical Considerations. There are several considerations when deploying CoinShuffle++ in practice. First, Bitcoin charges transactions with a small *fee* to prevent DoS attacks. Second, the mixing amount β must be the same for all peers but peers typically do not hold the exact mixing amount in their input Bitcoin account. Finally, after honestly performing the CoinShuffle++ protocol, a peer could spend her bitcoins in the input account before the CoinJoin transaction is confirmed, in an attempt of double-spending. All these challenges are easy to overcome. We refer the reader to the literature on CoinJoin based mixing, e.g., [40, 41, 54], for details on these practical considerations.

Compatibility. Since CoinJoin transactions work in the current Bitcoin network, CoinShuffle++ is immediately deployable.

4.4 Related Work

Privacy-preserving Currencies. Zerocoin [43] and its follow-up work Zerocash [10], whose implementation Zcash is currently in an alpha stage [2], are crypto-currencies protocols that provide anonymity by design. Although these solutions provide strong privacy guarantees, it is not clear whether Zcash will see widespread adoption, in particular given its reliance on a trusted setup due to use of zkSNARKS.

CoinShuffle++ builds on top of Bitcoin, and thus can be deployed immediately and seamlessly without requiring any changes to the Bitcoin protocol. Bitcoin is by far the most widespread crypto-currency and will most probably retain this status in the foreseeable future, so users are in need of solutions enhancing privacy in Bitcoin.

The CryptoNote design [60] relies on ring signatures to provide anonymity for the sender of a transaction. The advantage over CoinShuffle++ is that an online mixing protocol is not necessary; a sufficient anomyity set can be created using funds of user currently not online. However, this advantage comes with an important scaling drawback, because old transactions cannot be pruned from the blockchain (since anonymity ensures that it is not clear whether they have been spent or not). CoinShuffle++ does not suffer from this problem and is compatible with pruning spend transactions.

Centralized Mixing Services. Centralized mixing services [1] can be used to unlink a bitcoin from the bitcoin's owner: several owners transfer their coins to the mixing service who returns it to the owners at fresh addresses. The main advantage of a centralized approach it scales well to a large anonymity sets. However, by using these services, a user must fully trust the mix: First, anonymity is restricted towards external observers, i.e., the mixing service itself can still determine the owner of a bitcoin. Second and even more important, the users have to transfer their funds to the mixing service who could just steal them by refusing to give them back.

Mixcoin [12] mitigates the first problem by holding the mix accountable in case it steals the coins (but theft is still possible). Blindcoin [59] solves the second problem by the use of blind tokens.

Blindly Signed Contracts [30] proposes a centralized mechanism based on the combination of blind signatures and smart contract to solve both mentioned problems. However, the adoption of this approach requires a protocol change, which can be implemented as a soft-fork in the current Bitcoin blockchain. Moreover, this mechanism requires four Bitcoin transactions per peer, three of them to be confirmed sequentially. Even when using potentially risky one-block confirmations, this implies that mixing takes 30 minutes on average and transaction fees for four transactions per peer.

CoinShuffle++ uses a single transaction for all peers and thus requires much less time and fees from the peers in the mixing.

Other P2P Approaches. CoinParty [62] is a protocol where a set of mixing peers is used to mix coins from the users. In this approach, they assume that 1/3 of the mixing parties are honest. However, this trust assumption is not in line with the Bitcoin philosophy, and much worse, it is unclear how to realize it in a P2P setting without strong identities, where sybil attacks are easily possible. CoinShuffle++, instead, does not make any trust assumption on the mixing participants, except that there must be two honest peers, which is fundamental requirement for any protocol providing anonymity.

Sybil-Resistant Approaches. Xim [11] improves on its related previous work [9] in that it uses a fee-based advertisement mechanism to pair partners for mixing, and provides

Algorithm 3 CoinShuffle++

```
procedure MessageGen( )
    $(vk, sk) := \mathsf{AccountGen}()$                ▷ Stores $sk$ in the wallet
    return $vk$
procedure Confirm($i, P, my, \mathrm{VK}_{in}[\,], sk_{in}, \mathrm{VK}_{out}[\,], sid$)
    $tx := \mathsf{CoinJoinTx}(VK_{in}[], VK_{out}[], \beta)$
    $\sigma[my] := \mathsf{Sign}(sk_{in}, tx)$
    broadcast $\sigma[my]$
    receive $\sigma[\,]$ from all $p \in P$
        where $\mathsf{Verify}(\mathrm{VK}_{in}[p], \sigma[p], tx)$
    missing $P_{off}$ do                ▷ Peers refusing to sign are malicious
        return $P_{off}$
    $\mathsf{Submit}(tx, \sigma[\,])$
    return $\emptyset$                                ▷ Success!
```

evidence of the agreement that can be leveraged if a party aborts. Even in the simple case of a mixing between two peers, Xim requires to publish several Bitcoin transactions in the Bitcoin blockchain, what takes on average at least 10 minutes for each transaction.

CoinShuffle++ instead requires to submit a single transaction to the Bitcoin blockchain independently on the number of peers. Moreover, although CoinShuffle++ does not prevent malicious peers from disrupting the protocol, it provides a mechanism to identify the misbehaving peer so that it can be excluded and termination is ensured.

5. CREDIT MIXING IN RIPPLE

Almost all efforts to overcome the anonymity problem focus on crypto-currencies but ignore general settlement networks such as Ripple [8]. Nevertheless, a recent work [45] has shown that the lack of anonymity is a serious problem also in this scenario. The only available solution to this problem is a novel privacy-preserving protocol for credit networks [44]. This approach leverages the use of trusted hardware to enforce strong privacy guarantees by accessing the credit network by means of novel oblivious algorithms that hide the access pattern. This approach, however, is not fully compatible with the currently deployed Ripple network. Thus, we lack a protocol for privacy preserving settlement transactions that is fully compatible with Ripple.

In this section we first give the background on Ripple. Since Ripple is a generalization of a payment system, we aim at the same design goals as described for Bitcoin (Section 4.2). Then, we describe CreditMix, the first decentralized P2P mixing protocol for settlement networks. Finally, we discuss our experimental results from carrying out a CreditMix transaction in the real Ripple network.

5.1 The Ripple network

The Ripple network is a weighted, directed graph $\mathbb{G} := (\mathbb{V}, \mathbb{E})$, where $\mathbb{V}$ denotes the set of accounts and $\mathbb{E}$ represents the I Owe You (IOU) credit links between accounts. A weighted, directed edge $(u_1, u_2) \in \mathbb{E}$ is labeled with a dynamic scalar value α_{u_1,u_2} denoting the amount of *unconsumed* credit that u_1 has extended to u_2 (i.e., u_1 owes α_{u_1,u_2} to u_2). The available credit on an edge is lower-bounded by 0 and upper-bounded by ∞, although a tighter upper bound can optionally be adopted by the account owner. Additionally, every account has associated with it a non-negative amount of `XRP` (i.e., the native Ripple currency). For ease of explanation, we assume that there is only one IOU currency (i.e., USD) over the credit links in the Ripple network. We later discuss how CreditMix can handle several currencies.

As in Bitcoin, a Ripple account is created as a pair of signing and verification keys (vk, sk). The account is then labeled with an encoding of the hashed public key. A Ripple transaction contains a single sender and receiver. A transaction is valid when signed by the sender's private key.

Ripple defines two types of transactions: direct `XRP` payments and path-based settlement transactions. Intuitively, a direct payment involves the transfer of `XRP` between two accounts which may not have a credit path between them. Path-based settlement transactions settle credit between two accounts having a set of credit paths between them with enough capacity. In the following, we focus on settlement transactions and discuss direct `XRP` payments in Appendix B.

Assume that u_i wants to transfer β IOU to u_k and that u_i and u_k are connected through a path of the form $u_i - \ldots - u_j - \ldots - u_k$. In the path finding algorithm, links are considered as undirected. However, the transaction is performed by updating the credit on each link depending on its direction as follows: links in the direction from u_i to u_k are increased by β, while reverse links are decreased by β. A transaction is successful if no link is reduced to a value less than 0 and no link exceeds the pre-defined upper bound on the link (if other than ∞). A transaction can be split among several paths such that the sum of credit available on all paths is at least β. Such a transaction contains one sender and one receiver but several paths from sender to receiver.

A new account in the Ripple network needs to receive IOU on a credit link to interact with other accounts. The Ripple network solves this *bootstrapping* problem by introducing gateways. A gateway is a well-known reputed account that several accounts can trust to create and maintain a credit link in a correct and consistent manner. As gateways accounts are highly connected nodes in the Ripple network, the thereby created credit link will allow the new account to interact with the rest of the Ripple network. We briefly describe here the Ripple network. We refer to [8, 45] for more details.

5.2 Key Ideas

The main idea of our CreditMix protocol consists on leverage Fast-DC to mix n Ripple settlement transactions such that input and output accounts belonging to the same (honest) peer cannot be linked together. The problem of mixing n Ripple settlement transactions can be defined as follows:

Notation:

$(VK_{in}[i], sk_{in})$	Input account for user i
$(VK_{out}[i], sk_{out})$	Output account for user i
β	Amount of IOU to be mixed

Contract terms:

1. Every peer i has at least β IOU on her $VK_{in}[i]$'s link.
2. Every peer i has an account $VK_{out}[i]$ without IOU.
3. If no peer misbehaves, credit on each $VK_{in}[i]$ is decreased by β. Moreover, each user can send β credit from $VK_{out}[i]$.
4. If at least one peer misbehaves, credit on all accounts is maintained as defined on steps 1 and 2.

A Straw-man Direct Approach. A technical challenge is to ensure correct balance: no peer loses her IOU in the mixing process. A first trivial solution would be to let the peers shuffle their output accounts to obtain sender anonymity, agree in an ordering and perform n settlement transactions as follows: peer i transfers the mixing value to the output account in the position i within the shuffled list of output accounts. It is easy to see, however, that this approach fails to ensure the correct balance property: a peer that has already received her IOU in her output account can simply refuse to pay to the account assigned to her.

Adding a Shared Account. It is possible to create an account shared among the peers such that only when all peers agree, a transaction involving the shared account is performed. This effectively allows to add one synchronization round: each peer is forced to transfer β IOU to a shared account and only when $n \cdot \beta$ IOU are collected, they are sent to the output accounts. This, however, does not solve the correct balance problem either. Once all the IOU are collected in the shared account, a (malicious) peer could collaborate with the rest to create and sign the transaction to her output account and then disconnect. In this manner, IOU from the rest of peers are locked in the shared account.

Our Solution: Two Shared Accounts. The idea underlying our approach is to use two synchronization rounds via two shared accounts (e.g., vk^*_{in} and vk^*_{out}). In the first round, peers jointly create a credit link from each input account to vk^*_{in} with β IOU on them. Moreover, peers jointly create a credit link from each of the output accounts to vk^*_{out} with no IOU on them but an upper limit of β. At this point, credit at each $\mathrm{VK}_{out}[i]$ cannot be issued as part of a settlement transaction because vk^*_{out} does not have incoming credit yet (see Figure 4a). The second synchronization round can be then used to overcome that. All peers jointly create a transaction from vk^*_{in} to vk^*_{out} for a value of $n \cdot \beta$ IOU. Then, vk^*_{out} gets enough credit that can be used by each of the $\mathrm{VK}_{out}[i]$ (see Figure 4b).

5.3 Building Blocks

Ripple Network Operations. We utilize the following operations available in the current Ripple network.

$(vk, sk) := \mathsf{AccountGen}()$	Generate account keys
$tx := \mathsf{CreateTx}(vk_1, vk_2, \mathsf{v})$	Create settlement transaction
$tx := \mathsf{CreateLink}(vk_1, vk_2, \mathsf{v})$	Create link $vk_1 \to vk_2$ (limit v)
$tx := \mathsf{ChangeLink}(vk_1, vk_2, \mathsf{v})$	Modify link $vk_1 \to vk_2$ by v
$\{\mathsf{v}, \perp\} := \mathsf{TestLink}(vk_1, vk_2)$	Query link $vk_1 \to vk_2$
$\{0, 1\} := \mathsf{Submit}(tx, \sigma)$	Apply signed tx to network

A transaction tx becomes valid when is signed by the appropriate account's secret key. A tx from CreateTx and ChangeLink must be signed by sk_1, whereas a tx from CreateLink must be signed by sk_2.

In the following we assume for clarity of explanation that a tx is instantaneously applied to the Ripple network after invoking $\textsc{Submit}(tx, \sigma)$ with the correct signature. In practice, a tx is applied in a matter of seconds.

Digital Signature Scheme. Formally, we require the same digital signatures as described in Fast-DC (see Section 3.1).

Shared Account. For consistency with other protocols in the paper, we describe the handling of a shared account via multi-sign mechanism, even though it is not fully deployed in Ripple yet [48]. We discuss in Appendix A how to manage a shared account using distributed signatures in order to preserve full compatibility with the current Ripple.

A shared account is created as $vk^* := \mathsf{SAccountGen}(\mathrm{VK}[\,])$, where $\mathrm{VK}[\,]$ represents the vector of public keys for all peers jointly managing the shared account. The function $\{0, 1\} := \mathsf{SAccountVer}(vk^*, \mathrm{VK}[\,])$ allows peers to verify that the shared account has been correctly created. Then, each peer i can locally sign a tx involving the shared account as $\sigma_i := \mathsf{Sign}(sk_i, tx)$. A signature for a given tx is then verified as $\mathsf{Verify}(\mathrm{VK}[i], \sigma_i, tx)$. Finally, the tx is successfully submitted to the Ripple network by adding the correct signature from all peers as $\mathsf{Submit}(tx, (\sigma_1, \ldots, \sigma_n))$.

5.4 Our Protocol

CreditMix leverages Fast-DC and thus inherits all assumptions from it. Additionally, we assume that there is a leader peer (e.g., the first in the lexicographical order of their $\mathrm{VK}_{in}[\,]$), who is in charge of creating the shared accounts and broadcast them to the rest of peers. Finally, for ease of explanation we assume that every peer submits to the Ripple network a correctly signed transaction for shared accounts. This assumption does not have negative security implications: the transaction is only applied once since a Ripple transaction contains a sequence number to avoid replay attacks.

As in CoinShuffle++, for CreditMix we need to implement both $\textsc{MessageGen}()$ and $\textsc{Confirm}()$. The former is implemented by invoking $(vk, sk) := \mathsf{AccountGen}()$. The sk is stored for subsequent usage and vk is returned. The details for the latter are described in Appendix C. In the following, we give an overview of the operations through an example depicted in Figure 4. Assume that there are five peers with input accounts A^*_{in}, B^*_{in}, C^*_{in}, D^*_{in}, E^*_{in}, willing to shuffle $\beta = 10$ IOU. Further assume that all input accounts have at least β IOU available in the credit network. For the ease of explanation, we assume that all input accounts have a credit link with a common account (i.e., vk^*_{gw}). We discuss later in Appendix D how to relax this assumption. Given this setting, the protocol works as follows:

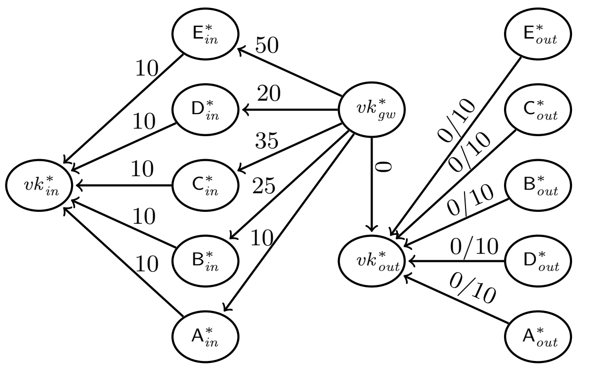
Phase 1: Create and Connect Input Shared Account. The leader uses the list of input verification keys VK_{in} to create the input shared account vk^*_{in} and broadcasts it to the rest of peers. To avoid the forgery of such message, the leader includes a signature on vk^*_{in} with the current round id i and session id sid. We require that only transactions starting at vk^*_{in} can be performed. For that, the *rippling* option must be disabled at each credit link for vk^*_{in} account. This *rippling* option is available in the current Ripple protocol and we refer the reader to [5] for details.

Then, peers jointly create a credit link from each input account $VK_{in}[i]$ to vk^*_{in}. For that, each peer locally creates a signature of the corresponding tx for each credit link and broadcasts the list of signatures to other peers. After verifying the correctness of all signatures, they are submitted to the Ripple network. In case some a peer creates a wrong signature, the protocol returns and consider him as misbehaving. Otherwise, each peer locally creates and signs a transaction that issues β credit to the recently created link $VK[i] \to vk^*_{in}$. If some peer refuses to fund such a credit link, returns him as a malicious peer.

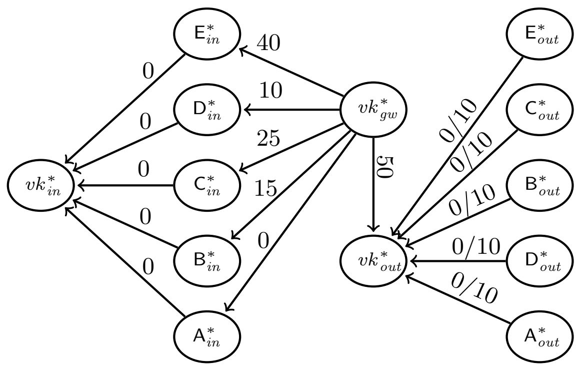
Phase 2: Create Credit Links for Output Accounts. As in the previous phase, the leader creates an output shared account (vk^*_{out}) and broadcasts it to the rest of peers. The account vk^*_{out} must enable transactions starting at it as well as transactions using vk^*_{out} as intermediate account. For that, *rippling* option must be enabled in this case. Then, peers jointly create a credit link from each $VK_{out}[i]$ to vk^*_{out} with an upper limit of β. These links allow them to anonymously perform a settlement transaction for up to β IOU.

The details of creating the links and verifying the corresponding signatures are similar to the previous case involving the input shared account. As before, peers ensure that only links to known output accounts are created. If during this phase some peer creates an invalid signature, the protocol is restarted without the misbehaving peer. Otherwise, the credit network at the end of this step is set up as depicted in Figure 4a.

Phase 3: Final Settlement Transaction. At this point, the vk^*_{out} account does not have any incoming credit and thus no transaction from an output account through vk^*_{out} can be performed. To solve this situation, the peers jointly create a settlement transaction transferring $n \cdot \beta$ IOU from vk^*_{in} to vk^*_{out}. This settlement transaction is possible using the n available paths through each of the peers' input accounts.



(a) Credit network after the set up of the shared accounts and output credit has been issued.



(b) Credit network after carrying out CreditMix without any disruptive peer.

Figure 4: An illustrative example of CreditMix protocol to mix 10 IOU among five peers. Solid arrows depict credit links between two accounts. Values a/b on the links denote: a current balance and b upper limit other than ∞. After a successful run of CreditMix, a user (e.g., Alice) can perform a settlement transaction for up to 10 IOU using $\mathsf{A}^*_{out}, vk^*_{out}, vk_{gw}$ as first hops in the payment path.

If some peer does not sign such transaction, the protocol is restarted without such peer.

Interestingly, this settlement transaction makes credit to flow from vk^*_{in} to vk^*_{out} so that the credit link between vk^*_{gw} and vk^*_{out} has now $n \cdot \beta$ IOU, as depicted in Figure 4b. This fact enables now settlement transactions from each output account to the rest of the credit network.

5.5 Security Analysis

Since CreditMix fulfills the application requirements defined in Section 3.2, sender anonymity in CreditMix is immediate from Fast-DC. In the following, we discuss how CreditMix enforces correct balance and termination.

Correct Balance. CreditMix ensures correct balance as follows. First, the creation and set up of the shared accounts do not involve the credit to be protected. Second, the deactivation of rippling option on vk^*_{in} credit links ensures that only settlement transactions starting at vk^*_{in} are accepted. This prevents a malicious peer from stealing honest peer's credit using vk^*_{in} as intermediate account (e.g., by means of a settlement transaction with path: $(\text{VK}[\text{malicious}]) - vk^*_{in} - (\text{VK}[\text{honest}]) - vk^*_{gw} - (\text{VK}[\text{malicious}])$.

Third, settlement transaction from vk^*_{in} to vk^*_{out} sends all the credit at once. Thus, either all peers contribute credit for the mixing process or none of them do. Moreover, this transaction is created and submitted to the Ripple network only if each peer has already an issuing transaction from vk^*_{out}. In this manner, it is ensured that credit in the output accounts can be used later to perform a transanction to any other account in the credit network.

The settlement transaction from vk^*_{in} to vk^*_{out} is the last step of the protocol. Thus, whenever the protocol is restarted due to a misbehaving peer, the credit on the links between the input accounts and vk^*_{gw} is not used and can be reused in another run of CreditMix. Finally, credit on the links between input accounts and vk^*_{in} can be locked after restarting. However, this credit is created only for the mixing purpose and it does not have any value outside the mixing protocol.

Termination. CreditMix achieves termination as follows. First, given the input to the protocol, every peer can generate the transactions to be signed. Second, the (possibly) randomized signatures of such transactions can be verified since the verification keys are available as input to the protocol. Thus, every peer can verify the correct behavior of each other peer and correctly detect a possible misbehavior.

5.6 Implementation

We have implemented the functionality for CONFIRM(...) defined in CreditMix. In order to maintain full compatibility with current Ripple and be able to test our approach in the currently deployed network, we have implemented the shared account management using a distributed signature scheme instead of multi-sign approach. In particular, we have implemented SAccountGen, SSign and Verify algorithms (see Appendix A) in JavaScript, based on the current Ripple code [50]. We have employed the EdDSA signatures based on Ed25519 curves.

Implementation-level Optimizations. First, both shared accounts vk^*_{in} and vk^*_{out} can be created in parallel. Second, links between vk^*_{in} and input accounts from peers can be created in parallel. Moreover, credit links from peers output accounts to vk^*_{out} can be created in parallel.

Testbed. In this test we measure the computation time required by each peer. We perform this test in a computer with Intel i7, 3.1 GHz processor and 16 GB RAM memory. Since peers needs to broadcast messages of size similar to those exchanged in Fast-DC, the communication time is similar to our results in Fast-DC.

Results. Given the aforementioned optimizations, we have studied the running time for a single run of SAccountGen and SSign algorithms. This thus simulates the creation of a single shared account and the signature of a transaction involving a shared account. We have observed that even with 50 participants, SAccountGen takes 1.215 seconds and SSign takes 0.306 seconds. It is important to note that it takes approximately 5 seconds for a transaction to be applied into the current Ripple network. Thus, the overall running time of CreditMix is mandated by the time necessary for the Submit operations.

A Real World Example. We have simulated a run of

CreditMix without disruption in the currently deployed Ripple network. As a proof of concept, we have successfully recreated the scenario depicted in Figure 4. Detailed information about the Ripple nodes[4] and the transactions[5] involved in the test can be found using the Ripple charts and RPC tools[6]. This demonstrates the compatibility of CreditMix with the Ripple network.

Other Practical Considerations. Our protocol CreditMix can perform privacy-preserving settlement transactions taken into account the rich functionality available in Ripple. For instance, CreditMix can handle transactions mixing different IOU currencies, fees associated to settlement transactions and allow mixing peers to have a credit link with different gateways. We discuss in detail these practical considerations in Appendix D.

6. OTHER APPLICATIONS OF FAST-DC

In this work we utilize Fast-DC to build anonymous transactions in currently deployed payment systems such as Bitcoin and Ripple. However, the techniques used in Fast-DC can be applied to improve upon currently available anonymous broadcast protocols.

Dissent [18] describes a shuffle protocol to perform a slot reservation that defines then a DC-net transmission schedule. In this manner, several peers can use their assigned slots to publish a message avoiding collisions. This shuffle protocol uses a number of communication rounds linear in the number of participants and requires that every peer sends a message of size quadratic in the number of participants. Fast-DC instead does not require a slot reservation protocol, uses only a constant number of communication rounds and messages are only of size linear in the number of participants.

Florian et al. [22] define a protocol to allow a set of peers to create new pseudonyms unlinkable to the old ones. Additionally, it is possible to verify that new pseudonyms belong to registered peers without revealing their identities. In order to achieve that, they use the shuffle protocol defined in CoinShuffle. This shuffle protocol requires a number of communication rounds linear in the number of participants. Replacing it with Fast-DC would improve the efficiency of the overall protocol since only constant number of rounds are then required.

7. CONCLUSIONS

In this work we present Fast-DC, a P2P mixing protocol based on DC-nets that enable participants to anonymously publish a set of messages ensuring sender anonymity and termination. Fast-DC avoids slot reservation and still ensures that no collisions occur, not even with a small probability. This results in Fast-DC requiring only 4 rounds independently on the number of peers, and $4 + 2f$ rounds in the presence of f misbehaving peers. We have implemented Fast-DC and showed its practicality to enable privacy preserving operations in several scenarios.

For instance, we use Fast-DC to implement CoinShuffle++, a practical decentralized coin mixing for Bitcoin. Our evaluation results show that CoinShuffle++ is a promising approach to ensure sender anonymity in Bitcoin requiring no change in the current Bitcoin protocol.

[4]http://tinyurl.com/zc3yu8l
[5]http://tinyurl.com/hb9722d
[6]https://ripple.com/build/ripple-info-tool/

Finally, we present CreditMix, the first mixing protocol for credit networks such as Ripple. CreditMix uses Fast-DC and requires only two extra synchronization rounds to mix credit network transactions independently of the number of mixing participants. CreditMix does not require any change to current credit networks and we show it by carrying out a proof of concept mixing payment in the currently deployed Ripple network.

8. REFERENCES

[1] Bitcoin Wiki: Mixing Services. https://en.bitcoin.it/wiki/Category:Mixing_Services.

[2] New alpha release: libzcash. https://z.cash/blog/new-alpha-release-libzcash.html.

[3] NXT crypto-currency. https://nxt.org/.

[4] Stellar network. https://www.stellar.org/.

[5] Understanding the "NoRipple" flag. https://ripple.com/knowledge_center/understanding-the-noripple-flag/.

[6] Androulaki, E., Karame, G. O., Roeschlin, M., Scherer, T., and Capkun, S. FC'13.

[7] Anonymity.Online (AN.ON). . https://anon.inf.tu-dresden.de/index_en.html, 2003. Accessed May 2016.

[8] Armknecht, F., Karame, G. O., Mandal, A., Youssef, F., and Zenner, E. Ripple: Overview and outlook. TRUST'15.

[9] Barber, S., Boyen, X., Shi, E., and Uzun, E. Bitter to better. how to make Bitcoin a better currency. FC'12.

[10] Ben-Sasson, E., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., and Virza, M. Zerocash: Decentralized anonymous payments from Bitcoin. S&P'14.

[11] Bissias, G., Ozisik, A. P., Levine, B. N., and Liberatore, M. Sybil-resistant mixing for Bitcoin. WPES '14.

[12] Bonneau, J., Narayanan, A., Miller, A., Clark, J., Kroll, J., and Felten, E. Mixcoin: Anonymity for Bitcoin with accountable mixes. FC'14. 2014.

[13] Bos, J., and den Boer, B. Detection of disrupters in the dc protocol. EUROCRYPT'89.

[14] Cash, D., Kiltz, E., and Shoup, V. The twin diffie-hellman problem and applications. *J. Cryptol. 22*, 4 (2009).

[15] Chaum, D. The dining cryptographers problem: Unconditional sender and recipient untraceability. *J. Cryptol. 1*.

[16] Chaum, D. Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. *Comm. ACM 4*, 2 (1981).

[17] Corrigan-Gibbs, H., Boneh, D., and Mazières, D. Riposte: An anonymous messaging system handling millions of users. IEEE S&P'15.

[18] Corrigan-Gibbs, H., and Ford, B. Dissent: Accountable anonymous group messaging. CCS '10.

[19] Dingledine, R., Mathewson, N., and Syverson, P. Tor: The second-generation onion router. USENIX Security'04.

[20] Dunning, L. A., and Kresman, R. Privacy preserving data sharing with anonymous id assignment.

Transactions on Information Forensics and Security 8, 2 (2013).

[21] Eidswick, J. A. A proof of newton's power sum formulas. *The American Mathematical off-monthly 75*, 4 (1968).

[22] Florian, M., Walter, J., and Baumgart, I. Sybil-resistant pseudonymization and pseudonym change without trusted third parties. WPES '15.

[23] Freire, E. S. V., Hofheinz, D., Kiltz, E., and Paterson, K. G. Non-interactive key exchange. PKC'13.

[24] Goel, S., Robson, M., Polte, M., and Sirer, E. G. Herbivore: A Scalable and Efficient Protocol for Anonymous Communication. Tech. Rep. 2003-1890, Cornell University.

[25] Goldschlag, D. M., Reed, M., and Syverson, P. Hiding Routing Information. In *Information Hiding: First International Workshop* (1996).

[26] Goldschlag, D. M., Reed, M. G., and Syverson, P. F. Onion Routing. *Commun. ACM 42*, 2 (1999).

[27] Golle, P., and Juels, A. Dining cryptographers revisited. EUROCRYPT'04.

[28] Gould, H. W. The Girard-Waring power sum formulas for symmetric functions and Fibonacci sequences, 1997. http://www.mathstat.dal.ca/FQ/Scanned/37-2/gould.pdf.

[29] Harreveld, T. Dining cryptographer networks, 2012.

[30] Heilman, E., Baldimtsi, F., and Goldberg, S. Blindly signed contracts: Anonymous on-blockchain and off-blockchain Bitcoin transactions.

[31] Holley, E. Earthport launches distributed ledger hub via Ripple. http://www.bankingtech.com/420912/earthport-launches-distributed-ledger-hub-via-ripple/, 2016.

[32] Invisible Internet Project (I2P). . https://geti2p.net/en/, 2003. Accessed May 2016.

[33] klinck (pseudonym). PARI-python interface. https://code.google.com/archive/p/pari-python/.

[34] Kosba, A., Miller, A., Shi, E., Wen, Z., and Papamanthou, C. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts, 2015. http://eprint.iacr.org/.

[35] Koshy, P., Koshy, D., and McDaniel, P. FC'14.

[36] Krasnova, A., Neikes, M., and Schwabe, P. Footprint scheduling for dining-cryptographer networks. FC'16.

[37] Liu, A. Ripple Labs Signs First Two US Banks. https://ripple.com/ripple-labs-signs-first-two-us-banks/.

[38] Liu, A. Santander: Distributed Ledger Tech Could Save Banks $20 Billion a Year. https://ripple.com/blog/santander-distributed-ledger-tech-could-save-banks-20-billion-a-year/, 2015.

[39] Maxwell, G. https://bitcointalk.org/index.php?topic=279249.msg2984051#msg2984051.

[40] Maxwell, G. CoinJoin: Bitcoin privacy for the real world. Post on Bitcoin Forum, 2013. https://bitcointalk.org/index.php?topic=279249.

[41] Meiklejohn, S., and Orlando, C. Privacy-enhancing overlays in bitcoin. BITCOIN'15.

[42] Meiklejohn, S., Pomarole, M., Jordan, G., Levchenko, K., McCoy, D., Voelker, G. M., and Savage, S. A fistful of bitcoins: Characterizing payments among men with no names. IMC'13.

[43] Miers, I., Garman, C., Green, M., and Rubin, A. D. Zerocoin: Anonymous distributed e-cash from bitcoin. S&P'13.

[44] Moreno-Sanchez, P., Kate, A., Maffei, M., and Pecina, K. Privacy preserving payments in credit networks: Enabling trust with privacy in online marketplaces. NDSS'15.

[45] Moreno-Sanchez, P., Zafar, M. B., and Kate, A. Linking wallets and deanonymizing transactions in the Ripple network. PETS'16.

[46] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system. Technical report, 2008.

[47] Reid, F., and Harrigan, M. An analysis of anonymity in the bitcoin system. Security and Privacy in Social Networks.

[48] Ripple. How to Multi-Sign. https://ripple.com/build/how-to-multi-sign/.

[49] Ripple. Reserves. https://ripple.com/build/reserves/.

[50] Ripple. Ripple implementation. https://github.com/ripple.

[51] Ripple. The Ripple protocol: A deep dive for financial professionals.

[52] Rizzo, P. Fidor Becomes First Bank to Use Ripple Payment Protocol. http://www.coindesk.com/fidor-becomes-first-bank-to-use-ripple-payment-protocol/, 2014.

[53] Rizzo, P. Royal Bank of Canada Reveals Blockchain Trial With Ripple. http://www.coindesk.com/royal-bank-canada-reveals-blockchain-remittance-trial-ripple/, 2016.

[54] Ruffing, T., Moreno-Sanchez, P., and Kate, A. CoinShuffle: Practical decentralized coin mixing for Bitcoin. ESORICS'14.

[55] Southurst, J. Australia's Commonwealth Bank Latest to Experiment With Ripple. http://www.coindesk.com/australia-commonwealth-bank-ripple-experiment/, 2015.

[56] Spagnuolo, M., Maggi, F., and Zanero, S. BitIodine: Extracting intelligence from the Bitcoin network. FC'14.

[57] Srikanth, T. K., and Toueg, S. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing 2*, 2 (1987), 80–94.

[58] Syta, E., Corrigan-Gibbs, H., Weng, S.-C., Wolinsky, D., Ford, B., and Johnson, A. Security analysis of accountable anonymity in Dissent. *TISSEC 17*, 1 (2014).

[59] Valenta, L., and Rowan, B. Blindcoin: Blinded, accountable mixes for Bitcoin. BITCOIN'15.

[60] van Saberhagen, N. Cryptonote. https://cryptonote.org/whitepaper.pdf.

[61] White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. An integrated experimental environment for distributed systems and networks. *SIGOPS 36* (2002).

[62] Ziegeldorf, J. H., Grossmann, F., Henze, M., Inden, N., and Wehrle, K. CoinParty: Secure multi-party mixing of bitcoins. CODASPY'15.

APPENDIX

A. HANDLING SHARED ACCOUNTS USING DISTRIBUTED SIGNATURES

A shared account can be managed through three algorithms: SAccountGen, SSign and Verify. Currently Ripple supports two digital signature algorithms: ECDSA using the curve SECP256k1 and EdDSA signatures using the elliptic curve Ed25519. For compatibility with the current Ripple network and for efficiency, we use the distributed signature scheme based on the EdDSA signature scheme. In the following, we describe SAccountGen and SSign algorithms in more detail. Verify works as defined in EdDSA.

```
procedure SAccountGen($VK_{in}[], my, P$)
    for all participant $p \in P \cup \{my\}$ do
        $VK^* := VK^* + \mathrm{VK}_{in}[p]$
    return $VK^*$
procedure SSign($P, my, VK_{in}[], sk_{in}, m$)
    $(vk_r, sk_r)$ := AccountGen()
    broadcast $vk_r$
    receive $vk_r[p]$ from all participant $p \in P$
    for all participant $p \in P \cup \{my\}$ do
        $vk_r^* := vk_r^* + vk_r[p]$
    $c := H(vk_r^*, vk_r^*, m)$
    $s := sk_r + c \cdot sk_{in}$
    broadcast $s$
    receive $s[p]$ from participant $p \in P$
    for all participant $p \in P \cup \{my\}$ do
        $s^* := s^* + s[p]$
    return $(vk_r^*, s^*)$
```

B. CreditMix: MIXING XRP

The XRP currency is defined in Ripple to protect the network from abuse and DoS attacks. A Ripple account needs to hold XRP for two reasons: the account is considered active only if it has a certain amount of XRP; moreover, the issuer of any transaction must pay a transaction fee in XRP.

The direct XRP payments allow the exchange of XRP between two accounts. Assume that u wants to pay β XRP to v and that u has at least β XRP in her XRP balance. Then β XRP are removed from u's XRP balance and added to v's XRP balance. Notice that this type of transaction does not require the existence of any (direct or indirect) credit line between the sender and the receiver of the transaction. Therefore, the *Path* field of the transaction is not used.

B.1 Key Ideas

The problem of mixing n XRP payment can be defined as a contract, analogously to mixing n settlement transactions (see Section 5). The contract can be defined as follows:

Notation:

$(VK_{in}[i], sk_{in})$	Input account for user i
$(VK_{out}[i], sk_{out})$	Output account for user i
β	Amount of XRP to be mixed

Contract terms:

1. Every peer i has at least β XRP on her $VK_{in}[i]$'s link.
2. If no peer misbehaves, XRP on each $VK_{in}[i]$ is decreased by β. Moreover, XRP on each $VK_{out}[i]$ is increased by β.
3. If at least one peer misbehaves, XRP on all accounts is maintained as defined on steps 1.

Given the similarity between XRP payments and bitcoin payments, we could ensure such contract by using a hypothetical CoinJoin transaction in Ripple, as we did for Bitcoin. However, at the time of writing, such transaction type is not available in Ripple. Thus, we devise a protocol that can be deployed in the current Ripple protocol without requiring any non-existing functionality.

Naively using two shared accounts. A first naive solution would consist in using two shared accounts to perform the mixing of n XRP payments in a similar manner to what we defined for shuffling n settlement transactions (see Section 5). There is however an important subtlety that appears when using XRP payments: making a payment from an input account to the vk_{in}^* implies actual sending of XRP. However, if when XRP are sent to vk_{in}^*, then a peer disconnects, the XRP in vk_{in}^* are stucked, disallowing the possibility of finishing the shuffling and other peers lose their XRP. Thus, ensuring correct balance is a challenge in this scenario.

Use of recover payments. It is possible to create in advance a payment from vk_{in}^* to a peer's input account. Then, the peer knows she can get her XRP back from vk_{in}^* if the mixing is not completed. Given that, she can safely send XRP from her input account's to vk_{in}^*. However, this does not completely solve the correct balance problem. When, the recover payment is created for the second peer, he can use it to steal the XRP from vk_{in}^* previously sent by the first peer.

Chaining and tagging recover payments. In Ripple, every transaction has a sequence number associated to it. For a given account, a transaction using this transaction as sender is only valid if its sequence number is one bigger than the last valid transaction. Given that, it is possible to ensure that recover payments are executed in order. The first peer gets a recover payment with sequence 1 and the second peer gets recover payment with sequence 2. This, however, does not totally solve the correct balance problem yet: the second peer can only recover his XRP if the first one submits here recover payment. Otherwise, all XRP are locked at vk_{in}^*.

We can fully solve the correct balance problem as follows. The first peer gets a recover payment with sequence 1 and tagged with her identifier (e.g., Alice). The second user gets two recover payments: one with sequence 1, tagged with his identifier (e.g., Bob) that returns Alice's XRP to Alice's account. A second one with sequence 2, tagged with Bob's identifier that returns his XRP. This mechanism ensures that Bob recovers his XRP even if Alice is going offline. Moreover, this mechanism ensures termination: given that transactions are tagged, a misbehaving peer maliciously recovering his XRP can be easily detected.

B.2 Building Blocks

Chaining of Ripple transactions. A Ripple transaction is only valid if the *Sequence number* field is set to a value 1 unit greater than the last-validated transaction from the same account. We use this to make sure that transactions issued from a certain account are in the expected order[7].

[7]https://ripple.com/build/transactions/#identifying-transactions

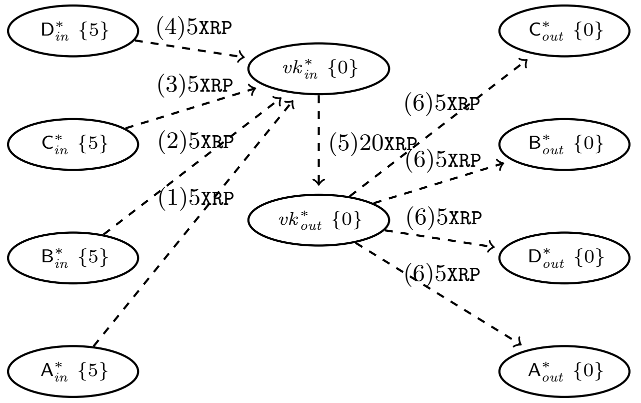


Figure 5: An illustrative example of CreditMix protocol to mix 5 XRP among 4 participants. Arrows depicts XRP transactions. The information above the arrows shows how many XRP are sent in every transaction and the number in () shows the sequence of the transactions. The last four transactions have the same sequence number since they can be performed in parallel after transaction 5. Numbers in {} represent the XRP balance at each account before performing the CreditMix protocol

In this protocol, similar to the CreditMix definition to mix IOU, we make use of anonymous broadcast and shared accounts as building blocks (see Section 5.3).

In the following, we describe the protocol steps by means of an example. For simplicity, assume that there are 4 participants with Ripple accounts A^*_{in}, B^*_{in}, C^*_{in}, D^*_{in}, willing to mix 5 XRP. Moreover, every participant knows her own output Ripple account. For example, Alice's output Ripple account is A^*_{out}. Given that setting, the protocol works as depicted in Figure 5. In detail:

1. **Create output list.** Participants gather together and perform the anonymous broadcast protocol to get a shuffled list of A^*_{out}, B^*_{out}, C^*_{out}, D^*_{out}.
2. **Create shared accounts.** The peers jointly generate two shared accounts vk^*_{in} and vk^*_{out}.
3. **Create transactions for (almost) the entire workflow.** The peers jointly create a transaction transferring 20 XRP from vk^*_{in} to vk^*_{out}. Moreover, they create a transaction sending 5 XRP from vk^*_{out} to every participant's out account. These transactions are all signed by all participants using the distributed signing algorithm. As vk^*_{in} and vk^*_{out} do not have any fund at this point in the protocol, transactions cannot be accepted to the ledger. Moreover, the sequence number of the transactions created at this step are known because the set of transactions to be performed are fixed.
   At this moment, as soon as every peer transfers her 5 XRP to vk^*_{in}, this account gets enough funds so that all the previous transactions are valid and they can be submitted by any peer to the Ripple network.
4. **Transfer input XRP to vk^*_{in} in a safe manner.** The main idea of this step is that every peer first creates a transaction to recover her coins from the vk^*_{in}. Once the transaction is correctly signed by every other peer, she sends her coins to vk^*_{in}. In detail:
   (a) Alice publishes a transaction sending 5 XRP from vk^*_{in} to A^*_{in}, tagged with "for Alice" and with sequence number 1. Bob, Carol and Dave send their share of the signature of the transaction to Alice.
   (b) Alice creates transaction sending 5 XRP from A^*_{in} to vk^*_{in} to the Ripple network. As the sender of this transaction is Alice, she can sign it on her own and send it. Moreover, note that Alice could submit the transaction created in previous step. However, other peers will see the tag and blame Alice of misbehaviour.
   (c) Bob creates two transactions. One transaction sending 5 XRP from vk^*_{in} to A^*_{in}, tagged as "for Bob" with sequence number 1. The second transaction sends 5 XRP from vk^*_{in} to B^*_{in} and is tagged as "for Bob" and has sequence number 2. Alice, Carol and Dave send their signature of the transactions to Bob.
   (d) Bob creates a transaction sending 5 XRP from B^*_{in} to vk^*_{in} to the Ripple network.
   (e) Carol publishes three transactions:
      - sender: vk^*_{in}, receiver:A^*_{in}, tag:"for Carol", seq# 1
      - sender: vk^*_{in}, receiver:B^*_{in}, tag:"for Carol", seq# 2
      - sender: vk^*_{in}, receiver:C^*_{in}, tag:"for Carol", seq# 3

      Alice, Bob and Dave send their signature of the transactions to Carol.
   (f) Carol sends the transaction sending 5 XRP from C^*_{in} to vk^*_{in} to the Ripple network.
   (g) Dave publishes four transactions:
      - sender: vk^*_{in}, receiver:A^*_{in}, tag:"for Dave", seq# 1
      - sender: vk^*_{in}, receiver:B^*_{in}, tag:"for Dave", seq# 2
      - sender: vk^*_{in}, receiver:C^*_{in}, tag:"for Dave", seq# 3
      - sender: vk^*_{in}, receiver:D^*_{in}, tag:"for Dave", seq# 4

      Alice, Bob and Carol send their signature of the transactions to Dave.
   (h) Dave sends the transaction transferring 5 XRP from D^*_{in} to vk^*_{in} to the Ripple network.
5. **Finish protocol if step 4 successful** vk^*_{in} will have the needed 20 XRP. Then, one of the peers can send the peers created and signed in step 3 and every output account receives the corresponding 5 XRP.
6. **Finish protocol if step 4 unsuccessful** If one of the peers refuses to send her XRP to vk^*_{in}, peers which already sent their coins can recover them with the recover payments created in step 4. For example, assume that Carol refuses to send her XRP to vk^*_{in}. Then, Bob can send his two recover payments. If Alice has sent already her recover payment, Bob can send only his second recover payment.
   The other case is that one of the peers sends his recover payments before the protocol is finished. Because every peer's recover payments are tagged with the peer's identifier itself, it is possible to detect him and blame him of misbehavior.

C. CreditMix PROTOCOL

We show the full pseudocode for CreditMix in Algorithm 4.

Algorithm 4 CreditMix

```
procedure CONFIRM($i, P, my, \mathrm{VK}_{in}[\,], sk_{in}, \mathrm{VK}_{out}[\,], sid$)
    $P_{all} := P \cup \{my\}$
    ▷ Leader: create shared accounts
    $vk^*_{in}$ := SAccountGen($\mathrm{VK}_{in}[\,]$)
    $vk^*_{out}$ := SAccountGen($\mathrm{VK}_{in}[\,]$)
    broadcast ($vk^*_{in}, vk^*_{out}$, Sign($sk_{leader}, (vk^*_{in}, vk^*_{out}, i, sid)$))
    ▷ Other peers receive and verify the shared accounts
    receive ($vk^*_{in}, vk^*_{out}, \sigma$) from leader
        where SAccountVer($vk^*_{in}, \mathrm{VK}_{in}[\,]$)
              ∧ SAccountVer($vk^*_{out}, \mathrm{VK}_{in}[\,]$)
              ∧ Verify($vk_{leader}, \sigma, (vk^*_{in}, vk^*_{out}, i, sid)$)
    missing $P_{off}$ do
        return $P_{off}$
    ▷ Create credit links $\mathrm{VK}_{in}[p] \to vk^*_{in}$
    for all $p \in P_{all}$ do
        $\mathrm{LINK}_{in}[p]$ := CreateLink($\mathrm{VK}_{in}[p], vk^*_{in}, \infty$)
        $\sigma_{in}[my][p]$ := Sign($sk_{in}, \mathrm{LINK}_{in}[p]$)
    broadcast ($\sigma_{in}[my][\,]$)
    receive ($\sigma_{in}[p][\,]$) from all $p \in P$
        where $\forall_{i \in P_{all}}$Verify($\mathrm{VK}_{in}[p], \sigma_{in}[p][i], \mathrm{LINK}_{in}[i]$)
    missing $P_{off}$ do
        return $P_{off}$
    ▷ Submit credit links $\mathrm{VK}_{in}[p] \to vk^*_{in}$
    for all $p \in P_{all}$ do
        Submit($\mathrm{LINK}_{in}[p], (\sigma_{in}[\,][p])$)
    ▷ Fund credit links $\mathrm{VK}_{in}[p] \to vk^*_{in}$
    $link'_{in}$ := ChangeLink($\mathrm{VK}_{in}[my], vk^*_{in}, \beta$)
    $\sigma'_{in}$ := Sign($sk_{in}, link'_{in}$)
    Submit($link'_{in}, \sigma'_{in}$)
    ▷ Verify $\mathrm{VK}_{in}[p] \to vk^*_{in}$ link for every participant
    $P_{inconsistent} = \emptyset$
    for all $p \in P$ do
        $v$ := TestLink($\mathrm{VK}_{in}[p], vk^*_{in}$)
        if $v = \perp \vee\ v < \beta$ then
            $P_{inconsistent} = P_{inconsistent} \cup \{p\}$
    if $P_{inconsistent} \neq \emptyset$ then
        return $P_{inconsistent}$
    ▷ Create credit links $\mathrm{VK}_{out}[p] \to vk^*_{out}$
    for all $p \in P_{all}$ do
        $\mathrm{LINK}_{out}[p]$ := CreateLink($\mathrm{VK}_{out}[p], vk^*_{out}, \beta$)
        $\sigma_{out}[my][p]$ := Sign($sk_{in}, \mathrm{LINK}_{out}[p]$)
    broadcast ($\sigma_{out}[my][\,]$)
    receive ($\sigma_{out}[p][\,]$) from all $p \in P$
        where $\forall_{i \in P_{all}}$Verify($\mathrm{VK}_{in}[p], \sigma_{out}[p][i], \mathrm{LINK}_{out}[i]$)
    missing $P_{off}$ do
        return $P_{off}$
    ▷ Submit credit links $\mathrm{VK}_{out}[p] \to vk^*_{out}$
    for all $p \in P_{all}$ do
        Submit($\mathrm{LINK}_{out}[p], (\sigma_{out}[\,][p])$)
    ▷ Create link $vk^*_{gw} \to vk^*_{out}$
    $link_{gw}$ := CreateLink($vk^*_{gw}, vk^*_{out}$)
    $\sigma_{gw}[my]$ := Sign($sk_{in}, link_{gw}$)
    broadcast ($\sigma_{gw}[my]$)
    receive ($\sigma_{gw}[p]$) from all $p \in P$
        where Verify($\mathrm{VK}_{in}[p], \sigma_{gw}[p], link_{gw}$)
    missing $P_{off}$ do
        return $P_{off}$
    ▷ Submit link $vk^*_{gw} \to vk^*_{out}$
    Submit($link_{gw}, (\sigma_{gw}[\,])$)
    ▷ Final settlement transaction
    $tx$ := CreateTx($vk^*_{in}, vk^*_{out}, (|P_{all}|) \cdot \beta$)
    $\sigma_{tx}[my]$ := Sign($sk_{in}, tx$)
    broadcast ($\sigma_{tx}[my]$)
    receive ($\sigma_{tx}[p]$) from all $p \in P$
        where Verify($\mathrm{VK}_{in}[p], \sigma_{tx}[p], tx$)
    missing $P_{off}$ do
        return $P_{off}$
    Submit($tx, (\sigma_{tx}[\,])$)
```

D. PRACTICAL CONSIDERATIONS FOR MIXING IN RIPPLE

In this section we discuss how CreditMix handles the different practical considerations to perform mixing settlement transactions in Ripple.

Handling different currencies. CreditMix supports the mixing of several currencies using features inherent to Ripple. In particular, it is possible to use existing market makers[8] to perform the necessary currency exchanges so that the payment from vk^*_{in} to vk^*_{out} is successfully carried out.

Handling fees. Every account in a path might charge some fee as a reward for allowing a settlement transaction. Thus, the amount of IOU received by the receiver might be lower than the amount sent by the sender. However, CreditMix requires that in the settlement transaction from vk^*_{in} to vk^*_{out} at least $n \cdot \beta$ IOU are received by vk^*_{out}.

Nevertheless, this is not a burden to deploy CreditMix in Ripple. Ripple allows to check in real time the fees associated to a given payment path. Therefore, it is possible to set the necessary IOU between every input account and vk^*_{in} so that at least $n \cdot \beta$ IOU are received by vk^*_{out} in the final settlement transaction performed in CreditMix.

Using several gateways. For easy of explanation, we have assumed that all input accounts have a credit link to a common gateway (i.e., vk^*_{gw}). However, input accounts might have only credit links available to different gateways (e.g., $vk^*_{gw_1}$ and $vk^*_{gw_2}$). In such case, mixing is still possible in CreditMix. The peers can jointly create a credit link between vk^*_{out} and one of the gateways (e.g., $vk^*_{gw_1}$). The settlement transaction from vk^*_{in} to vk^*_{out} it is possible provided that the Ripple network has enough liquidity between $vk^*_{gw_2}$ and $vk^*_{gw_1}$ and paths of the form $vk^*_{in} - VK_{in}[i] - vk^*_{gw_2} - vk^*_{gw_1} - vk^*_{out}$ are available.

We observe that although technically is possible to handle this scenario, in practice it demands a little more effort from the peers. Each peer needs to check in advance the fees charged in the (possibly longer) paths and accordingly set the credit links $\mathrm{VK}[i]_{in} \to vk^*_{in}$ so that in the end of the CreditMix protocol, enough credit is received by vk^*_{out}.

Funding of new accounts. Ripple applies reserve requirements to each new account in order to prevent spam or malicious usage [49]. At the time of writing, Ripple applies a base reserve of 20 `XRP` and an additional reserve of 5 `XRP` for each of the credit links associated to the account. The creation of this reserve can also be handled in CreditMix.

Shared accounts can be funded using any account belonging to the peers: each peer can send its corresponding share of `XRP` reserve to each of the shared accounts. However, fresh output account from a peer cannot be activated directly from the peer input account, as this would break the sender anonymity property we are after with CreditMix.

Instead, since `XRP` payments are similar to Bitcoin payments, we envision that it is possible to create a CoinJoin transaction to anonymously send necessary `XRP` from input to output accounts. Alternatively, we propose a more elaborated `XRP` mixing protocol (see Appendix B) which is less efficient but totally compatible with current Ripple protocol.

[8]In Ripple, a market maker is an account that accepts IOU in a certain currency in one of its credit links and exchange them into IOU in another currency available in another link.