Founding Digital Currency on Secure Computation

Karim El Defrawy
HRL Laboratories
3011 Malibu Canyon Rd
Malibu, CA 90265
kmeldefrawy@hrl.com

Joshua Lampkins[*]
Department of Mathematics, UCLA
Box 951555
Los Angeles, CA 90095
jlampkins@math.ucla.edu

ABSTRACT

Most current digital currency schemes and associated ledgers are either centralized or completely distributed similar to the design adopted by Bitcoin. Centralized schemes enable accountability, but leave the privacy of users' identities and transactions in the hands of one organization. Distributed schemes can ensure better privacy but provide little accountability. In this paper we design a privacy-preserving proactively-secure distributed ledger and associated transaction protocols that can be used to implement an accountable digital currency that inherits the ledger's privacy and security features. One of the main technical challenges that we address is dealing with the increase in ledger size over time, an unavoidable aspect as the currency spreads and the ledger is required to be maintained for a long time in the future. We accomplish this by reducing the distributed (secret-shared) storage footprint and the required bandwidth and computation for proactively refreshing the ledger to ensure long-term confidentiality and security.

1. INTRODUCTION

Despite the large number of digital currency schemes that have been proposed, e.g., [18, 19, 9, 10, 7], few have been implemented and adopted by a large number of users. Bitcoin [37] is currently the scheme that enjoys the widest adoption and is attracting the most attention. In existing digital currency designs either the scheme is decentralized, similar to Bitcoin and its replicas [1, 2], or centralized [18, 19, 9, 10, 7]. The completely decentralized nature of digital currencies such as Bitcoin prevents accountability and oversight from being effectively implemented. If a Bitcoin wallet's private keys are lost or stolen (a common occurrence lately [35]), there is no way to recover coins from such wallets. Thieves can hardly be identified when bitcoins are stolen. If bitcoins are used to sell illegal goods, it is very difficult for law enforcement to identify the vendor. On the other hand, a major concern with centralized digital currencies is that an institution issuing them can undermine user's privacy and is a single point of failure, in sharp contrast to decentralized schemes.

Stepping back, we observe that *most* (physical and digital) currencies today effectively exist in the form of a ledger. For example, when someone stores money in a bank account, the bank keeps a centralized ledger indicating how much that person owns. In the case of a fully distributed scheme such as Bitcoin, the distributed ledger is stored (and verified) by a large number of nodes scattered throughout the Internet. By striking a balance between the centralized and decentralized frameworks, we present the design of a privacy-preserving and proactively-secure distributed ledger that can be used to implement a digital currency in which: (1) The ledger is kept not by an individual server or entity, but by a large (possibly dynamic) group of *Ledger Servers* preferably owned/operated by different entities. (2) Computation on the ledger is performed without revealing secret values of account balances or the identities of users. (3) The ledger is stored such that an adversary would have to corrupt a significant fraction, e.g., approaching 1/3 or 1/2, of the servers in order to alter or even recover the ledger. (4) The ledger is periodically refreshed so that even if portions of it are obtained by an attacker, they cannot be used with other ones obtained in the future to uncover any transactions or ledger entries. Such a ledger and the functionalities required to implement a digital currency's necessary transaction protocols can be cast as a special case of secure multiparty computation (MPC).

A scheme satisfying the above properties can be used to instantiate a multi-organization, or multi-government issued currency where each Ledger Server is controlled by one of the organizations or governments. In the case of a multi-government currency, Ledger Servers would (ideally) be located in multiple countries to prevent any single government from shutting them down.[1]

Finally, we stress that it is critically important to determine how the ledger of a digital currency scheme can be stored and maintained efficiently for a long period of time. If the ledger is to contain a record of every transaction, its size will only increase with time. For example, at time of writing, the size of the Bitcoin ledger has increased from 7GB to 17GB over the past year.[2] The costs of storing the

[*]Work performed while at HRL Laboratories.

CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.
Copyright © 2014 ACM 978-1-4503-2957-6/14/11...$15.00.
http://dx.doi.org/10.1145/2660267.2660293.

[1]We note that the politics of how this ledger is deployed and regulatory issues concerning it are interesting questions but are outside the scope of this paper.
[2] `http://blockchain.info/charts/blocks-size`.

Bitcoin ledger could become prohibitive if Bitcoin ever sees widespread adoption as pointed out by some researchers [32].

In our digital currency scheme, the transaction data is initially secret-shared among the Ledger Servers. This could present long-term storage and maintenance problems, since secret sharing increases the total amount of memory required for storing data, and refreshing secret-shared data consumes a lot of bandwidth. We solve this problem by designing a new secure protocol (outlined in Section 3.4.3) for efficiently converting secret sharings into batch sharings [28] without reconstructing stored secrets; this reduces both the storage and bandwidth required for maintaining and refreshing ledger entries and data.

Outline: In Section 2 we discuss related work. In Section 3, we provide the technical details and assumptions of the design and operation of the new ledger and the transaction and accountability protocols and the digital currency scheme that can be built using it. In Section 4, we discuss some possible modifications to the scheme, specifically focusing on anonymity and increasing the threshold for tolerated corruption in ledger nodes. In the Appendices we provide a functionality, simulator, and security proof for the novel protocol that converts single-sharings to batch-sharings, and sketch a security proof for the entire digital currency scheme. In the full version [23], we provide performance analysis of some of the subprotocols to estimate the time required to perform transactions and the proactive refreshing of the ledger.

2. RELATED WORK

Research Proposals for Digital Currency Schemes: E-cash was initially introduced by Chaum [18] in 1982. In [18], the author describes how blind digital signatures could be used to issue electronic coins that could be anonymously spent at a merchant. In [19], this idea was applied using blind RSA digital signatures; [19] also addresses double-spending. E-cash schemes that provide the ability for some authority to revoke the anonymity of transactions are called *fair payment systems*, which were independently introduced in [43],[8], and [6]. The scheme in [43] uses RSA moduli, whereas [6] and [8] are based on discrete logarithms. In [34], the group signature scheme of [14] was altered to construct a group *blind* signature scheme, and implemented e-cash in a manner similar to schemes that used single-signer blind signatures. A publicly verifiable secret sharing (PVSS) scheme is presented in [40], and the author suggests e-cash as one possible application. The idea is to take the e-cash scheme of either [11] or [27] and share some of the data using the PVSS scheme. An e-cash scheme with multiple issuing authorities that uses blind threshold signatures was presented in [31]; any t out of n issuers can issue coins using the blind signatures. In theory, one could construct a digital currency scheme from an existing scheme by having multiple banks/coin issuers and requiring coins from a threshold number of issuers in order to spend. However, this would require some mechanism for ensuring that the issuers keep copies of the same ledger. In addition, the issuers must coordinate to ensure that a user cannot double-spend by requesting n coins from n issuers and spending $n/2$ coins at each of two merchants (since $n/2$ would be more than a threshold amount). Exploring such a scheme may be an area for future work.

Deployed Distributed Digital Currencies: Bitcoin's [37] success may be largely attributed to being the first implementation of a completely decentralized digital currency; the ledger of Bitcoin transactions is stored and verified by a group of "miners". The miners are rewarded for verifying transactions and preserving the integrity of the ledger with "mined" bitcoins and transaction fees provided by Bitcoin users. An adversary that controls less than 50% of the mining network will, with high probability, be incapable of causing inconsistencies in the ledger. However, a recent paper suggests that the corruption threshold for certain attacks is likely no more than 25% [26]. In the wake of Bitcoin's success, several other digital currencies have been implemented that use the same basic design as Bitcoin but add features to it, e.g., Litecoin [1] and Primecoin [2]. As these coins are marketed largely as alternatives to Bitcoin, they are sometimes referred to as "altcoins." Zerocoin [36] improves the anonymity of Bitcoin using zero-knowledge proofs. The case of Zerocoin notwithstanding, there have been minor differences in altcoin designs, typically varying the utilized proof-of-work scheme. Finally, [3] proposes to certify Bitcoin addresses to provide users with opt-in guarantees to send (and receive) bitcoins only to (from) certified users. The scheme in [3] also enables the control of creation of Bitcoin addresses by trusted authorities. Certified addresses prevent man-in-the-middle attacks and provide assurance that entities users are interacting with have been certified, though it is still possible to lose private keys of certified addresses or have them stolen and thus losing bitcoins associated with them.

Our Work Compared To Other Schemes: To our knowledge, this is the first digital currency proposal (which we call *Proactively-private Digital Currency, PDC*) that uses secure multiparty computation (MPC) to keep a secret-shared ledger among a (possibly dynamic) group of servers. In this paper, we mainly focus on a version that provides revocable (escrowed) anonymity; however, Section 4 describes how adding a couple of modifications can provide non-revoca
ble anonymity if desired. Most digital currency schemes (e.g., [19, 43, 6, 34]) do not hide the value being transmitted or the balances in users' accounts. PDC hides both transmission values and balances (unless suspicious activity is reported) while still preventing overdrawing. If the values stored at addresses were known to an adversary, the adversary could then target users with a large number of coins and try to compromise their personal computers to steal such coins. Furthermore, law enforcement often views transactions involving large amounts of cash as inherently suspicious, regardless of whether there is any evidence of illegal activity; PDC eliminates concerns about such scrutiny.

Our design allows de-anonymizing and freezing of suspect addresses, allows users to retrieve lost coins, and allows the Ledger Servers to implement analytics on the accounts and transactions without violating user privacy. Although schemes with revocable anonymity have been presented ([13, 12, 6]), PDC provides a unique incentive for those entrusted with the user's identity to preserve anonymity unless it is necessary to de-anonymize. In the schemes mentioned above, the entities that revoke anonymity are not involved with the transactions. In PDC, the Ledger Servers are responsible for processing transactions and for de-anonymizing malicious users. Since the Ledger Servers collect a transaction fee on each transaction, they will each possess a (presumably) large amount of the currency. If the Ledger Servers de-anonymize users without proper justification, this could make the public less inclined to use the currency, hence devaluing it. This devaluation would have a direct, negative, financial impact on the Ledger Servers.

3. PRIVACY-PRESERVING PROACTIVELY-SECURE DIGITAL CURRENCY

This section provides an overview of the operation of the ***Proactively-private Digital Currency (PDC)*** scheme and outlines some cryptographic preliminaries. The section also describes details of maintaining the ledger, protocols to perform individual operations required in financial transactions, protocols for enforcing accountability, and the main loop executed by the Ledger Servers. The section ends with a sketch of the security arguments for our protocols (more details in the Appendices).

3.1 Overview of the PDC Scheme

The main idea behind the PDC scheme is to design a digital currency in which the ledger of balances and transactions is kept by a group of *Ledger Servers* in secret-shared form. The group of servers can be dynamic if mobile secret sharing is used [41]. Apart from the Ledger Servers, there are *users* who own units of the currency, which we call *coins*, and who may want to transfer coins to other users. Each user may have multiple *addresses*, which are just public keys. The address serves the same purpose as an account number at a bank. To ensure that the identities of users can be recovered if malicious activity occurs, an *Identity Verification Server (IVS)* will be used to facilitate linking identities to addresses with (threshold) revocable anonymity. (We describe in Section 4 a completely anonymous version of PDC with no IVS.)

The operation of the PDC scheme using the proposed ledger proceeds as follows:

Initialization: When a user wants to create an address, the user provides her identity to an IVS. The IVS checks that the provided identity is the identity of a real person who can be located and be accountable if illegal activity is detected, similar to existing Bitcoin payment systems such as Coinbase. If the verification succeeds, the IVS sends back to the user a signature on the identity. The user then generates a private/public key pair, with the public key serving as her address. The user sends to each ledger server a request to initialize an address; the request includes the public key that serves as the address, a share of the identity, and a share of the IVS's signature on the identity. The Ledger Servers then invoke an MPC-based signature verification protocol *without* revealing the identity. This MPC-based signature verification protocol is only executed once per address and does not have to work in real time. If the signature is valid, the Ledger Servers add the address, with zero balance, to the balance ledger. (Details of initializing an address are given in Section 3.5.1.)

Sending and Receiving Coins: When a user wants to send coins to some address, she sends to each Ledger Server the sending address, the receiving address, and a share of each bit in the binary representation of the transaction value. The ledger servers use a secure comparison protocol to ensure that the user is not overdrawing the balance stored at the address; the comparison protocol does not reveal the value stored at the address or the value being transmitted. If the user is not overdrawing, the transaction value and a transaction fee (collected by the Ledger Servers) is subtracted from the sending address, and the transaction value is added to the receiving address. The transactional data is then stored in the transaction ledger. (Details of processing transactions are given in Section 3.5.2.)

Balance Checking: When a user wants to check the balance of an address or view recent transactions to/from that address, she sends a request to each of the Ledger Servers, and each replies with its share(s) of the requested value(s). (Details of balance checks are given in Section 3.5.3.)

Accountability: If some law enforcement or other agencies have evidence that an address is associated with illegal activity, it can send this evidence to each of the Ledger Servers, and so long as more than a sufficient (function of tolerated corruption threshold) number of the Ledger Servers agree, they can de-anonymize the address by sending their shares of the associated identity to said agency. The address may also be frozen pending legal action. In addition, PDC can implement retrieval of lost coins. (An overview of such accountability protocols are given in Section 3.6.)

Long Term Confidentiality: The ledger maintained by the Ledger Servers contains (possibly highly) sensitive financial data, so it is imperative to ensure secure, long-term storage of such data. To keep the ledger secure, the Ledger Servers will periodically perform a proactive refresh protocol on all data that has been secret-shared. One technical challenge that must be solved is dealing with the increase in ledger entries over time and reducing its storage footprint and the bandwidth and computation required for the proactive refresh. (Details of proactive refresh and how these challenges are solved are given in Section 3.4.3.)

3.2 Cryptographic Preliminaries

The parties involved in PDC are the n Ledger Servers $S_1, \ldots, S_n$, the Identity Verification Server(s), V, and the users; an individual user is denoted U. For the protocols used in this paper, the threshold of corruption for the Ledger Servers is $t < (1/3 - \epsilon)n$ for some positive constant ϵ. Operations occur over a finite field $\mathbb{Z}_p$ for some prime $p > 2^{\gamma+\kappa}$, where γ is the number of bits needed to represent the total amount of coins in circulation and κ is a security parameter.

We use both Shamir single secret sharing [42] and Franklin-Yung batch sharing [28]. To share a single secret s, a party selects a degree t polynomial that is uniformly random subject to $f(0) = s$ and sends to each server S_i the point $f(i)$. To batch-share a collection of secrets $s^{(1)}, \ldots, s^{(\ell)}$, a party selects a polynomial g of degree $d = \ell + t - 1$ such that $g(-k) = s^{(k)}$ for $k = 1, \ldots, \ell$ and $g(-\ell - k)$ is random for $k = 1, \ldots, t$, and then sends to each server S_i the point $g(i)$. We set the batch size ℓ to $n - 3t$.

For both single secret sharing and batch sharing, the Berle kamp-Welch algorithm [5] is used to interpolate polynomials whenever interpolation is needed in the proposed protocols.

We assume a public key signature and encryption scheme. The operations of encrypting using key K and signing using the private key corresponding to K are denoted Enc_K and Sig_K. By abuse of notation, we write Sig_V to denote signatures using party V's private key. We assume secure, authenticated channels between all parties, as well as an authenticated broadcast channel. In practice, a broadcast channel would likely be implemented via a broadcast protocol; our timing analysis in the full version [23] assumes $t + 1$

communication rounds for broadcast, which can be achieved using, e.g., the broadcast protocol in [29].

The PDC scheme requires the use of cryptographic protocols from the MPC literature. In particular, it requires a protocol for secret sharing (denoted Share), for generating secret sharings of random values (Random), for generating secret sharings of zero values (Zero), for multiplying shared values (Mult), for publicly opening sharings (Open), for proactively refreshing secret sharings (Refresh, see details in Section 3.4.3), and for secure comparison (Compare). Except for secure comparison, we require protocols that can perform these operations with both single-sharings and batch-sharings. The batch sharing versions of the protocols will be subscripted with B (i.e., the protocol for generating a batch sharing of all zeros is Zero_B). Any protocols that perform these tasks will work for PDC, so long as the corruption threshold for the protocols is not lower than t and the protocols are unconditionally secure.[3] In particular, we could use the comparison protocol of [24], the proactive refresh protocol of [4], and use [21] for the other protocols. There is no Zero protocol in [21], but the secret sharing protocol in [21] can be modified to do this. The protocols in [21] are for single secret sharings, but they can easily be modified to work with batch sharings.

3.3 Adversary Model

We assume a fully malicious adversary that corrupts the Ledger Servers in a Byzantine manner, i.e., corrupted Ledger Servers may behave arbitrarily. The adversary is mobile so any number of Ledger Servers may be corrupted over the course of the protocol. The adversary is limited in how many Ledger Servers may be corrupt at any one time as follows: The computation is divided into phases. The period between consecutive executions of the proactive refresh protocol is one phase; the refresh protocol itself is considered to be in both adjacent phases. The adversary is allowed to corrupt and decorrupt Ledger Servers at will, so long as no more than t Ledger Servers are corrupted in any phase. See section 3.4.3 for a discussion of how long term security is maintained against a mobile adversary.

We assume that the Identity Verification Server is always passively corrupted, but never maliciously corrupted; this is necessary to guarantee that users cannot generate addresses without providing valid identities. Any number of users may be statically corrupted, meaning that the adversary decides at the outset of the protocol which users she wants to corrupt, and cannot corrupt users during protocol execution.

3.4 Ledger Details

3.4.1 Overview of Ledger

We require two ledgers to be maintained by the Ledger Servers. The first ledger keeps track of the balance in each address, and the second records transaction history; these are called the *balance ledger* and *transaction ledger*, respectively.

For the balance ledger, an individual entry for Ledger Server S_i will be of the form $(A, D(i), b(i), c)$, where A is the address, $D(i)$ is a share of the identity associated with the address, $b(i)$ is a share of the current balance stored in the address, and c is a counter that keeps track of how many transactions have been sent *from* address A (not counting transactions *to* A). The counter c is used when transferring coins to prevent double-spending attacks.

[3]By "unconditionally secure," we mean that the protocols do not make any computational hardness assumptions.

Entries in the transaction ledger are stored in one of two formats. Initially, the transaction values are shared among the Ledger Servers using normal single secret sharing. Once the number of secret-shared ledger entries is at least ℓ, groups of ℓ secret sharings are converted into individual batch sharings of size ℓ. Batch sharing reduces the amount of data the Ledger Servers need to store, and considerably reduces the cost of proactively refreshing the transaction ledger. Details of the conversion from secret sharings to batch sharings are given in Section 3.4.3. The protocol that achieves such conversion is one of the main contributions of this paper and can be used in other secure computation contexts.

An individual secret-shared transaction ledger entry for Ledger Server S_i will be of the form $(A_{\text{from}}, A_{\text{to}}, c, B, s(i))$, where A_{from} is the sending address, A_{to} is the receiving address, c is a counter indicating that this was the c^{th} transaction from address A_{from}, B is the number of the block in which the transaction was processed, and $s(i)$ is a share of the amount sent. An individual batch-shared transaction ledger entry for Ledger Server S_i will be of the form $\left(\{(A_{\text{from}}^{(k)}, A_{\text{to}}^{(k)}, c^{(k)}, B^{(k)})\}_{k=1}^{\ell}, s(i)\right)$, where $(A_{\text{from}}^{(k)}, A_{\text{to}}^{(k)}, c^{(k)}, B^{(k)})$ is the data corresponding to the transaction value stored in batch location k in the polynomial s.

3.4.2 Ledger Server Synchronization and Communication

Most existing secret sharing and MPC protocols assume a synchronous network setting. Real computer networks, such as the Internet, are asynchronous; synchronous protocols can work in an asynchronous setting assuming loosely synchronized clocks and bounded message delays, as shown in [33]. Any institution operating a Ledger Server will (presumably) have ample technological resources to provide accurate clocks and bandwidth sufficient to ensure messages arrive in a timely manner. However, there is no way to guarantee even loose synchrony on the part of the clients that use PDC. To coordinate Ledger Servers with clients, data from clients will be collected into *blocks*. There is a fixed time interval, M, that defines the length of each block of data. Assuming the operation of the system begins at time zero, the data an individual Ledger Server receives between times 0 and M goes into the first block, and in general, the data received between times $(T-1)\cdot M$ and $T \cdot M$ goes into the T^{th} block. Once the Ledger Servers finish collecting data for an individual block, they broadcast the contents of their block to all the other Ledger Servers.

If one of the Ledger Servers receives a secret share of some data, it will need to be sure that a sufficient number of the other servers have received shares of the same data before processing that data. So when the Ledger Servers broadcast their blocks of data, each will look at the blocks of the other Ledger Servers to determine if enough of the shares have been received by the servers. If enough shares have been received, the Ledger Servers will process the data in a synchronous fashion.

We assume that the delay for transmission from user to Ledger Server is less than M (which we assume to be a couple of seconds). In the case that network latency or malicious intent causes the client's shares to arrive at different servers at different times, the delay may be enough that the shares are collected into the T^{th} block for some servers and the $(T+1)^{\text{th}}$ block for other servers. To account for this possibility, messages will not be processed in the first block in which they are broadcast. Instead, they will be carried over into the next block and re-broadcast. Messages cannot be retained indefinitely, so no message is broadcast in more than 2 (consecutive) blocks for an individual Ledger Server. If a Ledger Server broadcasts a message in two consecutive blocks, and there are still not enough shares from the other Ledger Servers in the second block, the message is deleted.

Due to the lack of synchrony between the users and the Ledger Servers, each protocol that the user engages in with the Ledger Servers is broken into two protocols: a user part and a server part. When the user wants to perform some action, she runs the user protocol. The servers collect messages from users as a part of the main protocol, the Ledger_Server_Loop described in Section 3.7. When the user's data is broadcast in a block, the servers then engage in the server part of the protocol. We use a subscript of U to denote the user part of the protocol and a subscript of S to denote the server part of the protocol. So for instance, when a user wants to initialize an address, the user runs $\mathsf{Initialize\_Address}_U$; once the Ledger Servers are ready to process the received data, they run $\mathsf{Initialize\_Address}_S$. We refer to the two protocols together as Initialize_Address.

3.4.3 Ensuring Long-term Confidentiality by Proactively Refreshing the Ledgers

In standard multiparty computation protocols, it is assumed that an adversary can corrupt no more than a fixed fraction of the parties performing the computation. However, it is more realistic to assume that *a sophisticated adversary could eventually corrupt every party given a long enough period of time.* In the proactive security model, an adversary can corrupt any number of parties, but can only corrupt a fixed fraction in any given time; such an adversary is called a mobile adversary in [38]. Since safeguarding the ledger of a digital currency is highly critical, we argue that one should design the ledger to be secure in the proactive security model.

Protocols for proactively refreshing shared secrets proceed in two phases. In the first phase, a secret s shared with a polynomial P (such that $P(0) = s$) is updated by setting $P \leftarrow P + Q$, where Q is a polynomial of the same degree as P which is random subject to the constraint that $Q(0) = 0$.[4] Thus if an adversary learns no more than a threshold number of shares for the old P, this will give her no information about the secret when combined with shares of the new P, because the shares are independently distributed. In the second phase, parties who had previously been corrupted and may have lost or corrupted data (either due to alteration of memory by malware or by a hard reboot to remove malware) recover the lost shares by interacting with the other parties.

There are multiple proposals for proactive refresh schemes [30, 41, 4]. We provide an instantiation of PDC with the scheme from [4], as it only assumes secure channels, whereas the other two schemes use the discrete logarithm assumption. Although in practice secure channels would be implemented with a public key infrastructure (PKI), we would like to assume only a generic PKI instead of discrete logarithm-based PKI. When refreshing the ledger we utilize the protocol from [4] (called Block-Redistribute) with a threshold of $t < (1/3 - \epsilon)$ and a batch size of $\ell = n - 3t$, but we do not use player virtualization, and we do not require that ℓ be a power of two as in [4]. Although the protocol in [4] was designed for refreshing batch sharings, it can easily be modified to refresh secret sharings as well.

[4]If P is a batch sharing, then Q will be a sharing of a batch of all zeros.

Recall that all entries in the transaction ledger are initially secret-shared. Before we describe how the Ledger Servers coordinate proactive refresh, we first present a new protocol for converting secret sharings into batch sharings, Convert_Sharings, which is one of the main contributions of this paper. We note that since the ledger is only expected to grow in size, converting secret-shared values into batched secret-shared values saves storage requirements and speeds up the proactive-refresh process.

The following protocol takes as input a group of polynomials that share secrets in batches of size ℓ and outputs a group of polynomials that share the same secrets in batches of size ℓ'. In our PDC scheme, this protocol will be used to convert Shamir sharings (i.e., sharings with $\ell = 1$) into sharings of batch size greater than 1. For this protocol, we use the notation $[X]$ to denote the set of integers from 1 to X. Thus the Cartesian product $[X] \times [Y]$ is $\{(x, y) : 1 \le x \le X,\ 1 \le y \le Y\}$. The protocol Convert_Sharings uses a set $\mathcal{C}orr$ to keep track of which parties *may* be corrupt. In step 10, when one party accuses another, the parties make a worst-case assumption that *both* parties are corrupt. Note that this set is distinct from any set of disputes which may be used in the MPC sub-protocols.

Convert_Sharings

Parties: $S_1, \ldots, S_n$.

Input: Each Ledger Server holds shares of batch sharing polynomials $\{H_a^{(k)}\}_{(a,k)\in[\ell']\times[K]}$ of batch size ℓ.

Output: Each Ledger Server holds shares of batch sharing polynomials $\{V_b^{(k)}\}_{(b,k)\in[\ell]\times[K]}$ of batch size ℓ' that contain the same secrets as did the $H_a^{(k)}$.

1. Set $\mathcal{C}orr = \emptyset$.

2. The servers use Random_B to generate polynomials $H_a^{(K+1)}$ of degree $d = \ell + t - 1$ for $a \in [\ell']$. (These polynomials are used for masking and will be discarded later.)

3. Each S_i selects polynomials $\{U_i^{(k)}\}_{k\in[K+1]}$ of degree $d' = \ell' + t - 1$ such that $U_i^{(k)}(-a) = H_a^{(k)}(i)$ for all $(a,k) \in [\ell'] \times [K+1]$ and shares them via Share_B.

4. The servers invoke Random K times to generate K sharings of random values $\{r^{(k)}\}_{k\in[K]}$.

5. The servers invoke Open K times to publicly reveal $\{r^{(k)}\}_{k\in[K]}$.

6. Define $\widetilde{H}_a$ and $\widetilde{U}_i$ for $(a, i) \in [\ell'] \times [n]$ by

$$\widetilde{H}_a = \textstyle\sum_{k=1}^{K} r^{(k)} H_a^{(k)} + H_a^{(K+1)}$$
$$\widetilde{U}_i = \textstyle\sum_{k=1}^{K} r^{(k)} U_i^{(k)} + U_i^{(K+1)}.$$

Each server locally computes their shares of these polynomials.

7. Each server sends *all* their shares of $\widetilde{H}_a$ and $\widetilde{U}_i$ to *each* other server for each $(a, i) \in [\ell'] \times [n]$. (Note that this is *not* done using broadcast.)

8. Each server uses Berlekamp-Welch on the shares of $\widetilde{U}_i$ received in the previous step to interpolate $\widetilde{U}_i(-a)$ for each $(a, i) \in [\ell'] \times [n]$.
9. Each server uses Berlekamp-Welch on the shares of $\widetilde{H}_a$ to interpolate $\widetilde{H}_a(i)$ for each $(a, i) \in [\ell] \times [n]$.
10. Each S_j checks if $\widetilde{U}_i(-a) = \widetilde{H}_a(i)$ for each $(a, i) \in [\ell'] \times [n]$. If this does not hold for some $\widetilde{U}_i$, then S_j broadcasts $(S_j, \texttt{J'accuse}, S_i)$. All servers add S_i and S_j to $\mathcal{C}orr$. (After a server is added to $\mathcal{C}orr$, any further accusations from that server are ignored.)
11. Each server erases all their shares of $\widetilde{H}_a$ and $\widetilde{U}_i$, shares of $H_a^{(k)}$ for each $(a, k) \in [\ell'] \times [K + 1]$, and shares of $U_i^{(K+1)}$ for each $i \in [n]$.
12. Define G to be the set of the first $n - 2t$ servers *not* in $\mathcal{C}orr$. Let $\{z_1, \ldots, z_{n-2t}\}$ denote the set of indices of servers in G. Let $\lambda_{b,i}$ denote the Lagrange coefficients for interpolating the evaluation at $-b$ of a degree-d polynomial from the shares of servers in G (i.e., for a polynomial f of degree $\leq d$, $f(-b) = \sum_{m=1}^{n-2t} \lambda_{b,m} f(z_m)$).
Each Ledger Server locally computes its share of $V_b^{(k)} = \sum_{m=1}^{n-2t} \lambda_{b,m} U_{z_m}^{(k)}$ for each $(b, k) \in [\ell] \times [K]$.

To see that the interpolation in step 12 results in batch sharings of the correct values, note that $V_b^{(k)}(-a) = \sum_{m=1}^{n-2t} \lambda_{b,m} U_{z_m}^{(k)}(-a) = \sum_{m=1}^{n-2t} \lambda_{b,m} H_a^{(k)}(z_m) = H_a^{(k)}(-b)$ for each $(a, b, k) \in [\ell'] \times [\ell] \times [K]$. Thus the sharings $V_b^{(k)}$ hold the same secrets as $H_a^{(k)}$.

The Convert_Sharings protocol is used in the Ledger_Server _Loop (see Section 3.7) to convert single secret sharings to batch sharings as needed to conserve space.

The Ledger Servers use the protocol Refresh_Ledge to proactively refresh their ledger secret sharings and batch sharings.

Refresh_Ledger

Parties: $S_1, \ldots, S_n$.
Input: Each Ledger Server use all its ledger entries as input.
Output: Each Ledger Server holds refreshed shares of the shared data in the ledger.

1. Each Ledger Server broadcasts a list of all addresses that they have stored in the balance ledger. Any address that is broadcast by at least $n - t$ of the servers is added to a list S_A of addresses.
2. Each Ledger Server broadcasts a list of all transaction ledger entries whose sending and receiving addresses are in S_A (although they do not broadcast their shares of the transaction values). Any transaction broadcast by at least $n - t$ of the servers is added to a list S_T of transactions.
3. For each address in S_A and each secret-shared transaction in S_T, the Ledger Servers locally add the associated secret sharings into a list, $(f^{(1)}, \ldots, f^{(m_1)})$. Similarly, the batch-shared transactions in S_T are added into a list $(g^{(1)}, \ldots, g^{(m_2)})$.
4. The Ledger Servers invoke Refresh with $(f^{(1)}, \ldots, f^{(m_1)})$ as input and Refresh$_B$ with $(g^{(1)}, \ldots, g^{(m_2)})$ as input.
5. Each server adds its refreshed shares to the balance and transaction ledgers.

Since refreshing the ledger adds to the computational and communication cost of maintaining the system, the transaction ledger entries need not be retained indefinitely. They could be deleted after some fixed amount of time.

3.5 Transaction Protocols

This section describes the protocols used to initialize addresses, transfer coins from one address to another, and to retrieve data from the ledger of balances and transactions.

3.5.1 Initializing an Address

The user, U, has an Identity Verification Server, V, verify that the user's ID is legitimate, the user generates a private key and corresponding address, then the ID is secret-shared among the Ledger Servers. This allows the Ledger Servers to de-anonymize an address if needed at some point in time.

Each message sent to the Ledger Servers will contain a nonce. The nonce will contain both a time-stamp (used to prevent replay attacks) and a κ-bit random number.

Initialize_Address$_U$

Parties: U, V, $S_1, \ldots, S_n$.
Input: U's identifying information, ID.
Output: Signed messages m_i (described in the last step) to each S_i.

1. U sends ID to V.
2. V verifies that ID is the identity of a real person. If so, V returns $\text{Sig}_V(ID)$ to U. If not, V sends the message `reject` to U and the protocol terminates.
3. U generates a private key and corresponding public key A.
4. V generates polynomials f and g of degree t such that $f(0) = ID$ and $g(0) = \text{Sig}_V(ID)$ and sends $m_i = (\texttt{init\_addr}, nonce, A, V, f(i), g(i))$ along with $\text{Sig}_A(m_i)$ to each S_i.

As discussed in Section 3.4.2, after U runs Initialize_Address$_U$, the Ledger Servers collect the received data into blocks the Ledger_Server_Loop, and after broadcasting these blocks, they run Initialize_Address$_S$.

Initialize_Address$_S$

Parties: $S_1, \ldots, S_n$.
Input: A, V; at least $n - t$ of the S_i hold v_i, w_i. (The values v_i, w_i are the (purported) shares $f(i), g(i)$ received from the user in Initialize_Address$_U$.)
Output: Each S_i has a balance ledger entry for A.

1. The servers invoke Random twice to generate degree t polynomials r_1 and r_2, and each S_i that had input v_i, w_i broadcasts $v_i + r_1(i)$ and $w_i + r_2(i)$.
2. Each server performs the Berlekamp-Welch algorithm to determine if the values broadcast in the previous step lie on two degree t polynomials, and if so compute the constant terms. If the Berlekamp-Welch algorithm fails for either polynomial, the protocol terminates without initializing the address.

3. Let v denote the constant term interpolated from the $v_i + r_1(i)$, and let w denote the constant term interpolated from the $w_i + r_2(i)$. Each S_i locally computes its share of $f'(i) = v - r_1(i)$. This defines a new polynomial f' such that $f'(0) = f(0) = ID$, where f is the polynomial from $\mathsf{Initialize\_Address}_U$ (if U is honest). Similarly, the servers define and compute shares of $g'(x) = w - r_2(x)$ such that $g'(0) = g(0) = \mathrm{Sig}_V(ID)$.
4. The servers run an MPC protocol for verifying that the value of $\mathrm{Sig}_V(ID)$ received from U is, in fact, a valid signature for ID. This is performed using f' and g' as inputs. If the signature is invalid, the protocol terminates.
5. The servers invoke Zero to generate a polynomial b, and each S_i adds $(A, f'(i), b(i), 0)$ to the balance ledger.

The use of an MPC protocol to verify a secret-shared signature for a secret-shared identity is the slowest part of the protocol, but note that this process need only be performed once for each address. The complexity of the protocol will vary depending on which public key scheme is being used. Protocols from [20] for modular arithmetic on secret-shared values can be used for implementing that step.

3.5.2 Transfer

To transfer coins from one address to another, the sender secret-shares the amount to be transferred, the Ledger Servers use a secure comparison protocol to check that the sender is not overdrawing the address, and then the amount is subtracted from the sender's balance and added to the recipient's balance. However, it is also necessary to verify that the amount being transferred falls into the proper range; namely, it must be less than 2^γ. Therefore, instead of the sender secret sharing the value as a single field element, the sender will secret-share each of the γ bits of the value separately.

The transfer protocol must prevent replay attacks in which the adversary re-sends the transfer request, thereby moving more coins out of the sender's address than anticipated. Therefore, the (signed) transaction will contain a counter; the counter will be set to j for the j^{th} transaction out of the address. If the user forgets the value of the counter, she can perform a balance check (described later).

We allow the Ledger Servers to collect a transaction fee, z, for each transaction. For simplicity, we assume there is a single address, X, to which all fees are sent, although one could modify the protocol to divide the fees among the Ledger Servers if desired.

$\mathsf{Transfer}_U$

Parties: U, $S_1, \ldots, S_n$.
Input: Sending address A_1, receiving address A_2, and transfer value, s.
Output: Signed messages m_i (described in the last step) to each S_i.

1. U decomposes s into its binary representation, $(s^{(\gamma-1)}, \ldots, s^{(0)}) \in \mathbb{Z}_p^\gamma$, where $s^{(\gamma-1)}$ is the most significant bit and $s^{(0)}$ is the least significant bit.
2. U constructs γ degree t polynomials $f^{(\gamma-1)}, \ldots, f^{(0)}$ such that $f^{(k)}(0) = s^{(k)}$ for each $k = 0, \ldots, \gamma - 1$.
3. U sends $m_i = (\mathtt{tx}, nonce, A_1, A_2, j, (f^{(\gamma-1)}(i), \ldots, f^{(0)}(i)))$ and $\mathrm{Sig}_{A_1}(m_i)$ to each S_i.

After receiving the requests to transfer coins from the user, the Ledger Servers run their portion of the protocol.

$\mathsf{Transfer}_S$

Parties: $S_1, \ldots, S_n$.
Input: A_1, A_2, j; at least $n - t$ of the S_i hold $(v_i^{(\gamma-1)}, \ldots, v_i^{(0)})$. (The value $v_i^{(k)}$ is the (purported) share $f^{(k)}(i)$ received in $\mathsf{Transfer}_U$.)
Output: Each Ledger Server has a new entry in the transaction ledger; additionally, the balance ledger entries for A_1 and A_2 are updated to reflect the transaction.

1. The Ledger Servers invoke Random γ times to generate polynomials $(r^{(\gamma-1)}, \ldots, r^{(0)})$, and each S_i that had input $(v_i^{(\gamma-1)}, \ldots, v_i^{(0)})$ broadcasts $v_i^{(k)} + r^{(k)}(i)$ for each $k = 0, \ldots, \gamma - 1$.
2. Each server performs the Berlekamp-Welch algorithm to determine if the values broadcast in the previous step lie on γ degree t polynomials, and if so compute the constant terms. If the Berlekamp-Welch algorithm fails for any polynomial, the protocol terminates without transferring the coins.
3. Let $C^{(k)}$ denote the constant term interpolated from the $v_i^{(k)} + r^{(k)}(i)$. Each S_i locally computes its share of $g^{(k)}(i) = C^{(k)} - r^{(k)}(i)$ for each $k = 0, \ldots, \gamma - 1$. This defines new polynomials $g^{(k)}$ such that $g^{(k)}(0) = f^{(k)}(0) = s^{(k)}$, where the $f^{(k)}$ are the polynomials from $\mathsf{Transfer}_U$ (if U is honest).
4. The servers invoke Zero γ times to generate polynomials $\{\mu^{(k)}\}_{k=0}^{\gamma-1}$, invoke Mult to generate a sharing of $(g^{(k)} + \mu^{(k)} - 1)g^{(k)}$, and invoke Open on the sharing of the product for each $k = 0, \ldots, \gamma - 1$. If any of the opened values is not zero, the protocol terminates.
5. Each server locally computes its share $s(i) = \sum_{k=0}^{\gamma-1} 2^k g^{(k)}(i)$ of the degree t polynomial s.
6. Let b_1 and b_2 denote the polynomials stored in the balance ledger that represent the balances in the addresses A_1 and A_2 (respectively). The servers invoke Compare to verify that $b_1(0) \geq z + s(0)$. If this does not hold, the protocol terminates.
7. The servers locally update their shares of the balances in addresses A_1 and A_2 as follows: $b_1(i) \leftarrow b_1(i) - s(i) - z$ and $b_2(i) \leftarrow b_2(i) + s(i)$.
8. Each server locally updates the balance ledger entry for the address X, incrementing its share by z.
9. Each server locally increments the transaction counter in the balance ledger for address A_1 by one.
10. Each S_i adds the entry $(A_1, A_2, j, B, s(i))$ to the transaction ledger, where B is the block in which this transaction was processed.

3.5.3 Balance Checks and Transaction Confirmations

In the balance check protocol, the user sends a request for the data, and each Ledger Server sends its copy of the transaction "meta-data" (addresses, counter numbers, and block numbers) as well as shares of the required values. As far as secret sharings are concerned, this is relatively simple: Each Ledger Server sends the share as-is. For batch-shared transactions, some computation by the Ledger Servers is required before sending shares.

Suppose that U has requested some transactional data that is stored in a batch with other users' data. If U's requested data is stored in locations $k_1, k_2, \ldots, k_m$ within the batch, the Ledger Servers construct a *canonical sharing* of a batch with ones in locations $k_1, k_2, \ldots, k_m$ and zeros elsewhere. By canonical sharing, we mean a polynomial g such that $g(-k_j) = 1$ for each $j = 1, \ldots, m$ and $g(-k) = 0$ for all other $k = 1, \ldots, t + \ell$. Since a canonical sharing for a known set $\{k_1, k_2, \ldots, k_m\}$ is completely deterministic, the Ledger Servers can compute their shares locally without any interaction. This sharing is then added to a random sharing of a batch of all zeros for privacy reasons. The Ledger Servers invoke Mult_B to multiply this sum by the batch sharing in the transaction ledger. The shares of the resultant product are then sent to the user.

The user looks up transactions that she authorized by the counter number of the transaction. However, she has no way to know what the counter number for transactions she receives will be. Therefore, she looks them up by block number, which is equivalent to looking them up by the time at which the coins were received.

$\mathsf{Check\_Balance}_U$

Parties: U, $S_1, \ldots, S_n$.

Input: Address A, for which U holds the private key.

Output: U holds values of the requested ledger entries.

1. If U wants to view some transactions with A as the sending address, then U sets j_1 to be the earliest transaction counter value requested and j_2 to be the latest. Otherwise, U sets $j_1 = j_2 = \perp$. If U wants to view some transactions with A as the receiving address, then U sets B_1 to be the earliest block number requested and B_2 to be the latest. Otherwise, U sets $B_1 = B_2 = \perp$.
2. U sends $m = (\texttt{check\_bal}, nonce, A, j_1, j_2, B_1, B_2)$ along with $\mathrm{Sig}_A(m)$ to each server S_i.
3. U waits some pre-determined amount of time for the Ledger Servers to respond. Upon receiving shares of the requested data, U uses the Berlekamp-Welch algorithm to reconstruct the shared data.

After receiving balance check requests from users in the main loop, the Ledger Servers run their portion of the protocol.

$\mathsf{Check\_Balance}_S$

Parties: U, $S_1, \ldots, S_n$. **Input:** A, j_1, j_2, B_1, B_2.

Output: Requested ledger entries are sent to U.

1. Each server looks for the entry for address A in the balance ledger, all entries in the transaction ledger with A as the *sending* address with counter values between j_1 and j_2, and all entries in the transaction ledger with A as the *receiving* address with block numbers between B_1 and B_2. (If j_1 or j_2 equals $\perp$, then the server assumes that U does not want information on transactions sent from A, and similarly for B_1, B_2.)
2. If none of the data requested is batch-shared, then skip to step 6. Otherwise, let $f^{(1)}, \ldots, f^{(w)}$ denote all the batch-sharings that contain requested data.
3. Let $K^{(j)} = \{k_1^{(j)}, \ldots, k_{m_j}^{(j)}\}$ denote the locations in batch $f^{(j)}$ that correspond to U's data, and let $g^{(j)}$ denote the canonical sharing of the batch with ones in locations $K^{(j)}$ and zeros elsewhere.
4. The servers invoke Zero_B w times to generate polynomials $r^{(1)}, \ldots, r^{(w)}$ and locally compute their shares of $h^{(j)} = g^{(j)} + r^{(j)}$ for each $j = 1, \ldots, w$.
5. The servers invoke Mult_B to multiply each pair $(f^{(j)}, h^{(j)})$, resulting in sharings $s^{(j)}$ for $j = 1, \ldots, w$.
6. The servers send their shares of all the requested secret sharings to U, along with their shares of each $s^{(j)}$ if any batch-shared data was requested. Additionally, the servers send U the associated transaction meta-data and the current transaction counter in the balance ledger entry for A.

3.6 Accountability Protocols

One of the advantages of secret sharing the identity of the users is to prevent accidental or malicious loss of coins. This is a considerable problem with deployed schemes such as Bitcoin as they exist today. Bitcoin users often loose their private keys, and hence their coins. There have been several bitcoin thefts, in one case totaling \$1.2 million [35]. Even though the addresses to which the stolen coins are sent are often public knowledge, there is no means by which to return the coins to the victim or at least freeze the suspect address until the issue is resolved [25]. In theory, addresses could be frozen and transactions reversed in decentralized schemes, but this would require coordination among a decentralized group of ledger operators who have disparate interests and frequently operate anonymously. Each of the above problems can be easily addressed in PDC. We briefly describe solutions to these problems in this section (see the full version [23] for more details).

3.6.1 Address Freezing and De-anonymization

Suppose that some law enforcement or investigative entity, R, has reason to believe that a particular address, A, is associated with criminal or illegal activity. So long as more than $2t$ Ledger Servers agree that the address is suspect, the secret-shared identity can be revealed, and the Ledger Servers can refuse to process further transactions from/to that address. This provides some level of privacy guarantee to the user, in that the user knows her address cannot be de-anonymized unless there is some consensus.

The protocol for freezing operates as follows: Each Ledger Server that receives a properly signed message of the form $(\texttt{freeze}, nonce, A, E)$ from R examines the evidence, E, and determines whether or not the address, A should be frozen and de-anonymized. Each Ledger Server that agrees to cooperate with R its share of the identity to R. If R receives more than $2t$ shares, R interpolates the identity $D(0)$ of the user associated with address A. Each Ledger Server that decides that address A is suspect will ignore any further transaction requests that send coins to or from that address.

3.6.2 *Lost Coin Retrieval*

Since the private key associated with an address is used to sign all messages sent to the Ledger Servers, loosing the private key is tantamount to loosing the coins stored in the address, as the coins can no longer be accessed. Loss of private keys has been a persistent problem for Bitcoin users, as there is no way to retrieve such lost coins. Because the user's identity is linked to the address in PDC, one can retrieve lost coins.

Suppose the user has lost the private key for address L and wants to send the coins in L to a new address A. She constructs a degree t polynomial Q that is uniformly random subject to the constraint that $Q(0) = ID$, where ID is her identity. The user sends $(\texttt{ret\_coins}, nonce, L, A, Q(i))$ signed by the identity verification server, IVS, to S_i for each $i = 1, \ldots, n$. The Ledger Servers verify IVS' signature then subtract the sharing Q provided by the user from the sharing of the identity in the balance ledger entry for L. This sharing is then multiplied by a random number, and the product is publicly opened. If the opened value is zero, this means that the identity provided by the user is the same as the secret-shared identity stored in the balance ledger, so the Ledger Servers transfer the coins in L to A as requested.

3.7 Ledger Server Loop

The main protocol run by the Ledger Servers continues in an indefinite loop, since we assume the currency will exist for a long time. The core of the PDC scheme is the collection of data into blocks and the broadcasting of those blocks to the other servers. Different actions are taken according to what is found in those blocks. The servers are loosely synchronized so as to broadcast their blocks at approximately the same time. We refer to these points in time as *broadcast points*. We refer to the block broadcast by server S_i at broadcast point T as $B_i^{(T)}$.

Each message received by the Ledger Servers consists of a header and a collection of shares (although the collection of shares may be empty). These parts are denoted [*header*] and [*shares*], respectively. (For example, the [*header*] part of the message m_i sent in Initialize_Address$_U$ is $(nonce, A, V)$ and the [*shares*] part is $(f(i), g(i))$). Each header contains a nonce, and each nonce contains a time-stamp. Recall from Section 3.4.2 that the Ledger Servers assume that the delay for transmission from the user is less than M. A time-stamp is considered *out of date* if it varies from the Ledger Server's clock by at least M.

```
Ledger_Server_Loop
Ledger Server $S_i$ performs the following steps:
1. Set $T = 1$.
2. Set $B_i^{(T)} = \emptyset$. If $T \geq 2$, add all messages in $B_i^{(T-1)}$ that were not in $B_i^{(T-2)}$ to $B_i^{(T)}$.
3. Collect all properly formed and signed messages received from users, adding them to the set $B_i^{(T)}$ as they are received. Messages that are out of date, messages that are duplicates of messages already in $B_i^{(T)}$, and tx messages with incorrect transaction counters are deleted. (Duplicates are defined to be messages with the same nonce, init_addr messages with the same address, or tx messages with the same sending address.)
4. When the broadcast point $T$ is reached (at time $T \cdot M$), broadcast the headers of all messages in $B_i^{(T)}$.
5. If some [header] is broadcast by at least $n-t$ Ledger Servers, and if this is the second broadcast containing messages with that header, then one of steps 6 through 8 is performed:
6. If [header] = $(\texttt{init\_addr}, nonce, A, V)$, and if the address $A$ does not already exist in the balance ledger, then run Initialize_Address$_S$ with $A, V$ as public input and $S_i$ using the [shares] = $(v_i, w_i)$ portion of its message as private input.
7. If [header] = $(\texttt{tx}, nonce, A_1, A_2, j)$, and if the addresses $A_1, A_2$ already exist in the balance ledger, then run Transfer$_S$ with $A_1, A_2, j$ as public input and $S_i$ using the [shares] = $(v_i^{(\gamma-1)}, \ldots, v_i^{(0)})$ portion of its message as private input.
8. If [header] = $(\texttt{check\_bal}, nonce, A, j_1, j_2, B_1, B_2)$, and if the address $A$ already exists in the balance ledger, then run Check_Balance$_S$ with $A, j_1, j_2, B_1, B_2$ as public input. (Here, [shares] = $\emptyset$.)
9. If there are $m \geq \ell$ secret sharings in the transaction ledger, the Ledger Servers run Convert_Sharings on the first $\lfloor m/\ell \rfloor \ell$ of these sharings.
10. Set $T \leftarrow T + 1$.
11. Jump to step 2.
```

Note that we have not added the protocol Refresh_Ledger to the main loop. It could either be added between steps 10 and 11 in the main loop, or the Ledger Servers could run a separate loop in parallel with the main loop, constantly refreshing shared data.

3.8 Security Analysis

The PDC protocols described in this paper build upon protocols from the MPC literature (listed in Section 3.2) that are already proven secure in the Universal Composability (UC) framework in their respective papers, so security of our protocols can be proven using the UC composition theorem [17]. Since the share conversion protocol, Convert_Sharings, is one of the main technical contribution of this paper, we provide its functionality, simulator, and security proof in Appendix A. We sketch a proof of security for the entire PDC scheme in Appendix B. (Due to space constraints, we cannot provide a full security analysis and proof for each protocol in this version of the paper. Such details will be provided in the full version [23])

4. INCREASING ANONYMITY GUARANTEES AND CORRUPTION THRESHOLD

Increasing Anonymity Guarantees: PDC can be made strictly anonymous, without the user ever revealing her identity to anyone. The Identity Verification Server can be eliminated entirely, and the addition of an address to the balance ledger can be performed in essentially the same way as Bitcoin: The user generates a private key and corresponding public key as an address, and the Ledger Servers add this address to the balance ledger the first time that a user sends value to it. Note that lost coins cannot be restored using the protocol above in this case.

Anonymizing Traffic: We sketch a protocol allowing a group of k users to transfer funds from k addresses to k addresses without any Ledger Server learning which input address maps to which output address.

Let k be some power of 2. Then k users will send transfer requests to the Ledger Servers as specified in the Transfer protocol, except that the receiving address will be secret-shared. The transfer values will be subtracted from the sending addresses. Then each Ledger Server in turn will apply a permutation (unknown to the other Ledger Servers) to the secret-shared transfer values, and apply the same permutation to the secret-shared receiving addresses. This prevents any one Ledger Server from mapping input to output address.

A secret permutation is applied by server S_q as follows: S_q picks a random permutation π, and the Ledger Servers invoke PermuteLayer from the full version of [4] (found in refrence [2] in that paper) twice with the same permutation, once the sharings of the transfer values, and once on the sharings of the receiving addresses. Since the batch size ℓ in this case is 1, the only subprotocol used will be PermuteBetweenBlocks. Since the permutation is not publicly known, the Ledger Servers do not known their shares of f_I and $f_{\overline{I}}$ without input from S_q. So S_q generates polynomials f_I and $f_{\overline{I}}$ as needed. The Ledger Servers need to prove that one of these polynomials stores a zero and the other stores a one. They do this by opening the sum of the sharings and checking that it is one, and by invoking a secure multiplication protocol on the sharings and verifying that the product is zero. The same polynomials f_I and $f_{\overline{I}}$ will be used for both invocations of PermuteLayer to ensure that the same permutation is used.

Once the permutation is completed, the receiving addresses are publicly opened. Then the (permuted) transfer values are added to the receiving addresses.

Increasing the Corruption Threshold: Although our PDC scheme assumes signatures, the underlying protocols that manipulate secret sharings make no computational assumptions. Hence the threshold is $t < (1/3 - \epsilon)n$, which is near optimal for information theoretic protocols. The threshold can be raised to $t < (1/2 - \epsilon)n$ if some computational assumptions are made. For instance, instead of using "standard" secret sharing, the modified PDC could use Pedersen's scheme [39], which uses commitments based on the discrete logarithm assumption. Modifying the PDC sub-protocols to use Pedersen commitments is fairly straightforward.

5. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their helpful comments and our shepherd Abhishek Jain for his suggestions on how to address them. We also thank Joshua Baron, Jonathan Katz and Gene Tsudik for insightful discussions about digital currency and MPC.

6. REFERENCES

[1] Litecoin, 2013. `https://litecoin.org/`.

[2] Primecoin: Cryptocurrency with prime number proof-of-work, 2013. `http://primecoin.org/static/primecoin-paper.pdf`.

[3] Giuseppe Ateniese, Antonio Faonio, Bernardo Magri, and Breno de Medeiros. Certified bitcoins. Cryptology ePrint Archive, Report 2014/076, 2014. `http://eprint.iacr.org/`.

[4] Joshua Baron, Karim El Defrawy, Joshua Lampkins, and Rafail Ostrovsky. How to withstand mobile virus attacks, revisited. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 293–302, New York, NY, USA, 2014. ACM.

[5] Elwyn R. Berlekamp. *Algebraic Coding Theory*. Aegean Park Press, 1984.

[6] Ernest F. Brickell, Peter Gemmell, and David W. Kravitz. Trustee-based tracing extensions to anonymous cash and the making of anonymous change. In *SODA*, pages 457–466, 1995.

[7] J. Camenisch, A. Lysyanskaya, and M. Meyerovich. Endorsed e-cash. In *Security and Privacy, 2007. SP '07. IEEE Symposium on*, pages 101–115, 2007.

[8] Jan Camenisch. Group signature schemes and payment systems based on the discrete logarithm problem, 1998.

[9] Jan Camenisch, Susan Hohenberger, and Anna Lysyanskaya. Compact e-cash. In *EUROCRYPT 2005*, pages 302–321. 2005.

[10] Jan Camenisch, Susan Hohenberger, and Anna Lysyanskaya. Balancing accountability and privacy using e-cash. In *Security and Cryptography for Networks*, pages 141–155. 2006.

[11] Jan Camenisch, Ueli M. Maurer, and Markus Stadler. Digital payment systems with passive anonymity-revoking trustees. In *ESORICS*, pages 33–43, 1996.

[12] Jan Camenisch, Ueli M. Maurer, and Markus Stadler. Digital payment systems with passive anonymity-revoking trustees. *Journal of Computer Security*, 5(1):69–90, 1997.

[13] Jan Camenisch, Jean-Marc Piveteau, and Markus Stadler. An efficient fair payment system. In *ACM Conference on Computer and Communications Security*, pages 88–94, 1996.

[14] Jan Camenisch and Markus Stadler. Efficient group signature schemes for large groups. In Jr. Kaliski, BurtonS., editor, *Advances in Cryptology CRYPTO '97*, volume 1294 of *Lecture Notes in Computer Science*, pages 410–424. Springer Berlin Heidelberg, 1997.

[15] R. Canetti. Universally composable signature, certification, and authentication. In *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE*, pages 219–233, June 2004.

[16] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[17] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. *IACR Cryptology ePrint Archive*, 2000:67, 2000.

[18] David Chaum. Blind signatures for untraceable payments. In *CRYPTO*, pages 199–203, 1982.

[19] David Chaum, Amos Fiat, and Moni Naor. Untraceable electronic cash. In *CRYPTO*, pages 319–327, 1988.

[20] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for

equality, comparison, bits and exponentiation. In *TCC*, pages 285–304, 2006.

[21] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In *CRYPTO*, pages 572–590, 2007.

[22] Ivan Damgård, Yuval Ishai, and Mikkel Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In Henri Gilbert, editor, *Advances in Cryptology EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 445–465. Springer Berlin Heidelberg, 2010.

[23] Karim El Defrawy and Joshua Lampkins. Founding digital currency on secure computation (full version of this paper). use your favorite search engine to find it.

[24] Peter Bogetoft et al. Secure multiparty computation goes live. In *Financial Cryptography*, pages 325–343, 2009.

[25] Sarah Meiklejohn et al. A fistful of bitcoins: Characterizing payments among men with no names. *IMC*, 2013.

[26] Ittay Eyal and Emin Gun Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *Financial Cryptography*, 2014.

[27] Yair Frankel, Yiannis Tsiounis, and Moti Yung. "indirect discourse proof" : Achieving efficient fair off-line e-cash. In *ASIACRYPT*, pages 286–300, 1996.

[28] Matthew K. Franklin and Moti Yung. Communication complexity of secure computation (extended abstract). In *STOC*, pages 699–710, 1992.

[29] Juan A. Garay and Yoram Moses. Fully polynomial byzantine agreement for $n >$ processors in $t + 1$ rounds. *SIAM J. Comput.*, 27(1):247–290, 1998.

[30] Amir Herzberg, Stanislaw Jarecki, Hugo Krawczyk, and Moti Yung. Proactive secret sharing or: How to cope with perpetual leakage. In *CRYPTO*, pages 339–352, 1995.

[31] Wen-Shenq Juang. A practical anonymous off-line multi-authority payment scheme. *Electronic Commerce Research and Applications*, 4(3):240–249, 2005.

[32] Dan Kaminsky. Some thoughts on bitcoin, 2011. `http://www.slideshare.net/dakami/bitcoin-8776098`.

[33] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In *TCC*, pages 477–498, 2013.

[34] Anna Lysyanskaya and Zulfikar Ramzan. Group blind digital signatures: A scalable solution to electronic cash. In *Financial Cryptography*, pages 184–197, 1998.

[35] Robert McMillan. $1.2m hack shows why you should never store bitcoins on the internet, 2013. `http://www.wired.com/wiredenterprise/2013/11/inputs/`.

[36] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *IEEE Symposium on Security and Privacy*, pages 397–411, 2013.

[37] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. `http://bitcoin.org/bitcoin.pdf`.

[38] Rafail Ostrovsky and Moti Yung. How to withstand mobile virus attacks (extended abstract). In *PODC*, pages 51–59, 1991.

[39] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO*, pages 129–140, 1991.

[40] Berry Schoenmakers. A simple publicly verifiable secret sharing scheme and its application to electronic. In *CRYPTO*, pages 148–164, 1999.

[41] David Schultz. *Mobile Proactive Secret Sharing.* PhD thesis, Massachusetts Institute of Technology, 2007.

[42] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[43] Markus Stadler, Jean-Marc Piveteau, and Jan Camenisch. Fair blind signatures. In *EUROCRYPT*, pages 209–219, 1995.

APPENDIX

A. CONVERT_SHARINGS SECURITY PROOF

We prove security of the protocol Convert_Sharings in the framework of Universal Composability (UC) [17]. A real world protocol is said to be UC-secure if an environment, $\mathcal{Z}$, that provides inputs and receives outputs from the protocol cannot distinguish whether it is interacting with the real (or hybrid) protocol or an ideal one that uses an ideal functionality performing the same task as the real world protocol. We compare the ideal protocol with a hybrid protocol which replaces calls to sub-protocols with calls to sub-functionalities. The ideal functionality $\mathcal{F}_{\mathsf{CS}}$ for Convert_Sharings is:

Description of $\mathcal{F}_{\mathsf{CS}}$.

1. *Input Phase*
   - *1.1* $\mathcal{Z}$ instantiates each server S_i with its input to the protocol, which is ℓ, ℓ', K, n, t and its share of the degree $d = \ell + t - 1$ polynomial $H_a^{(k)}$ for each $(a, k) \in [\ell'] \times [K]$. $\mathcal{Z}$ initializes the adversary $\mathcal{A}$ with auxiliary input z.
   - *1.2* All servers send the inputs they received from $\mathcal{Z}$ to $\mathcal{F}_{\mathsf{CS}}$. If inputs are inconsistent, $\mathcal{F}_{\mathsf{CS}}$ outputs (`abort`) and aborts.
2. *Corruption Phase*
   $\mathcal{A}$ may iteratively request to corrupt servers by sending messages $(\texttt{corrupt}, S_i)$ to $\mathcal{F}_{\mathsf{CS}}$. For each such corruption, $\mathcal{F}_{\mathsf{CS}}$ sends $H_a^{(k)}(i)$ to $\mathcal{A}$ for each $(a, k) \in [\ell'] \times [K]$ and sends $(\texttt{corrupt})$ to S_i. After each corruption, $\mathcal{A}$ may provide new input shares for S_i to $\mathcal{F}_{\mathsf{CS}}$.
3. *Output Phase*
   - *3.1* $\mathcal{F}_{\mathsf{CS}}$ sends $(\texttt{Shares?}, \ell, \ell', K)$ to $\mathcal{A}$.
   - *3.2* $\mathcal{A}$ sends $\mathcal{F}_{\mathsf{CS}}$ shares $V_b^{(k)}(i)$ for each corrupt S_i and for each $(b, k) \in [\ell] \times [K]$.
   - *3.3* $\mathcal{F}_{\mathsf{CS}}$ interpolates from the shares of $H_a^{(k)}$ received from the honest servers the points $H_a^{(k)}(-b)$ for each $(a, b, k) \in [\ell'] \times [\ell] \times [K]$.
   - *3.4* $\mathcal{F}_{\mathsf{CS}}$ chooses degree $d' = \ell' + t - 1$ polynomials $V_b^{(k)}$ uniformly at random, subject to the constraint that they agree with the shares received from $\mathcal{A}$ and they satisfy $V_b^{(k)}(-a) = H_a^{(k)}(-b)$ for each $(a, b, k) \in [\ell'] \times [\ell] \times [K]$.
   - *3.5* $\mathcal{F}_{\mathsf{CS}}$ provides outputs $V_b^{(k)}(i)$ to each honest S_i for each $(b, k) \in [\ell] \times [K]$. $\mathcal{F}_{\mathsf{CS}}$ provides out-

puts to the corrupt servers as specified by the adversary.

Note that the functionality does not include a post-output corruption phase or a post-execution corruption phase as in [16]. The reason for eliminating post-output corruption is that the adversary chooses its output, and hence learns no new information in the output phase. Without loss, we can assume that the adversary only corrupts after receiving new information. (This follows the convention used in [22].) Post-execution corruption is not needed when secure erasures are assumed, as stated in [16]. The reason is that messages sent and received during the protocol can be deleted after the protocol is finished. Thus the adversary has no information to learn from the protocol execution after the protocol has finished.

The ideal world adversary (simulator) is given below. The intuition behind it is as follows: The simulator begins by emulating the sub-functionalities as if generating sharings of polynomials $H_a^{(K+1)}$ and $U_i^{(k)}$ for $(i,k) \in [n] \times [K]$. In reality, the simulator simply provides to the hybrid-world adversary the shares it requested, and whenever the adversary requests to corrupt another party, the simulator chooses random shares. The only difficulty occurs in step 11. At this point, the simulator has already provided the adversary with *every* share of $\widetilde{H}_a$ and $\widetilde{U}_i$. Thus the shares that the simulator provides to the adversary upon corruption must match with what has already been sent. The key point here is that $\widetilde{H}_a$ is a sum of polynomials with $H_a^{(K+1)}$ as a summand, and $H_a^{(K+1)}$ only exists in the hybrid execution, not in $\mathcal{F}_{\mathsf{CS}}$. So upon corruption, the simulator forwards the shares of $H_a^{(k)}$ for $k \in [K]$ received from $\mathcal{F}_{\mathsf{CS}}$ to the adversary, and then chooses a share of $H_a^{(K+1)}$ so that the sum matches with the share of $\widetilde{H}_a$ that has already been sent. The process is similar for the $\widetilde{U}_i$.

Before describing the simulator, we need to describe the sub-functionalities used. The functionality $\mathcal{F}_{\mathsf{Share}_B}$ used in the hybrid model will be essentially the same as the functionality $\mathcal{F}_{\mathsf{RobustShare}}$ from [22]. The functionalities $\mathcal{F}_{\mathsf{Random}}$ and $\mathcal{F}_{\mathsf{Random}_B}$ will be essentially the same as $\mathcal{F}_{\mathsf{double}}$ from [22], except that only degree d (and no degree $2d$) sharings will be created, and for $\mathcal{F}_{\mathsf{Random}}$ the batch size is 1. Both $\mathcal{F}_{\mathsf{Share}_B}$ and $\mathcal{F}_{\mathsf{Random}_B}$ will be modified so that the batch size (ℓ or ℓ') is an input to the functionality.

The functionality $\mathcal{F}_{\mathsf{Open}}$ receives from all players the shares of the sharing(s) to be opened, interpolates the secret(s), and then sends the secret(s) to each player and to the adversary.

Simulator $\mathcal{S}$ for Convert_Sharings.

1. $\mathcal{S}$ internally runs $\mathcal{A}$ with the inputs (ℓ, ℓ', K) and the auxiliary input z that $\mathcal{S}$ received from $\mathcal{Z}$.
2. $\mathcal{S}$, emulating $\mathcal{F}_{\mathsf{Random}_B}$, sends $(\texttt{Shares?}, \ell', \ell)$ to $\mathcal{A}$ (where ℓ' represents the number of sharings being generated and ℓ represents the batch size used for those sharings).
3. $\mathcal{A}$ sends shares $H_a^{(K+1)}(i)$ to $\mathcal{S}$ for each corrupt S_i and each $a \in [\ell']$.
4. $\mathcal{S}$, emulating $\mathcal{F}_{\mathsf{Share}_B}$, sends $(\texttt{Shares?}, \{1,\ldots,n\}, n(K+1), \ell')$ to $\mathcal{A}$ (where $\{1,\ldots,n\}$ represents the identities of the dealers, $n(K+1)$ represents the number of sharings being generated, and ℓ' represents the batch size used for those sharings).
5. $\mathcal{A}$ sends shares $U_j^{(k)}(i)$ to $\mathcal{S}$ for each corrupt S_i and each $(j,k) \in [n] \times [K+1]$. $\mathcal{A}$ also sends polynomials $U_i^{(k)}$ of degree d' to $\mathcal{S}$ for each corrupt S_i and each $k \in [K+1]$.
6. $\mathcal{S}$, emulating $\mathcal{F}_{\mathsf{Random}}$, sends $(\texttt{Shares?}, K)$ to $\mathcal{A}$ (where K represents the number of sharings being generated).
7. $\mathcal{A}$ sends shares $R^{(k)}(i)$ to $\mathcal{S}$ for each corrupt S_i and each $k \in [K]$.
8. Emulating $\mathcal{F}_{\mathsf{Open}}$, $\mathcal{S}$ receives (again) from $\mathcal{A}$ the shares received in the previous step and sends uniformly random values $\{r^{(k)}\}_{k\in[K]}$ to $\mathcal{A}$.
9. $\mathcal{A}$ may iteratively request to corrupt servers. Each $(\texttt{corrupt}, S_i)$ message that $\mathcal{S}$ receives from $\mathcal{A}$ is forwarded to $\mathcal{F}_{\mathsf{CS}}$, and shares $H_a^{(k)}(i)$ that $\mathcal{S}$ receives from $\mathcal{F}_{\mathsf{CS}}$ are forwarded to $\mathcal{A}$.
10. $\mathcal{S}$ chooses degree d polynomials $\widetilde{H}_a$ that are random subject to the constraint that $\widetilde{H}_a(i) = \sum_{k=1}^{K} r^{(k)} H_a^{(k)}(i) + H_a^{(K+1)}(i)$ for each corrupt S_i. For each honest S_j, $\mathcal{S}$ chooses degree d' polynomials $\widetilde{U}_i$ that are random subject to the constraint that $\widetilde{U}_j(i) = \sum_{k=1}^{K} r^{(k)} U_j^{(k)}(i) + U_j^{(K+1)}(i)$ for each corrupt S_i and $\widetilde{U}_j(-a) = \widetilde{H}_a(j)$ for each $a \in [\ell']$. For each corrupt S_i, define $\widetilde{U}_i = \sum_{k=1}^{K} r^{(k)} U_i^{(k)} + U_i^{(K+1)}$ using the polynomials already received from $\mathcal{A}$.
11. Emulating the honest servers, $\mathcal{S}$ sends messages for the honest servers as in step 7 of Convert_Sharings, with $\mathcal{A}$ emulating the (recipient) corrupt servers. More specifically, $\mathcal{A}$ iteratively selects each honest server to send messages to dishonest servers in sequence, with $\mathcal{S}$ sending messages on behalf of the honest servers according to the protocol specification of Convert_Sharings, and and after each receipt of messages from an honest server, $\mathcal{A}$ may choose to iteratively corrupt servers. With each server corruption, the following steps are performed:
    - 11.1 $\mathcal{S}$ forwards the corruption request from $\mathcal{A}$ to $\mathcal{F}_{\mathsf{CS}}$.
    - 11.2 Upon receiving shares $H_a^{(k)}(q)$ for $(a,k) \in [\ell'] \times [K]$ of (newly) corrupt server S_q from $\mathcal{F}_{\mathsf{CS}}$, $\mathcal{S}$ computes shares $H_a^{(K+1)}(q) = \sum_{k=1}^{K} r^{(k)} H_a^{(k)}(q) - \widetilde{H}_a(q)$.
    - 11.3 $\mathcal{S}$ chooses uniformly random polynomials $U_q^{(k)}$ for $k \in [K]$ that are consistent with the shares already received from the adversary in step 5 and satisfy $U_q^{(k)}(-a) = H_a^{(k)}(q)$ for each $a \in [\ell']$.
    - 11.4 $\mathcal{S}$ computes $U_q^{(k+1)} = \sum_{k=1}^{K} r^{(k)} U_q^{(k)} - \widetilde{U}_q$. (Note that by way $\widetilde{U}_q$ was defined, this implies $U_q^{(K+1)}(-a) = H_a^{(K+1)}(q)$ for each $a \in [\ell']$.)
    - 11.5 $\mathcal{S}$ sends $H_a^{(k)}(q)$ for $(a,k) \in [\ell'] \times [K+1]$ and $U_q^{(k)}$ for $k \in [K+1]$ to $\mathcal{A}$.
12. After all honest servers have sent their shares in the previous step, the $\mathcal{A}$ sends the shares of the dishonest players to $\mathcal{S}$ according to the step 7 of Convert_Sharings.

13. $\mathcal{S}$ internally executes Convert_Sharings for each honest player up to step 10 at which point $\mathcal{S}$ and $\mathcal{A}$ send messages to each other according to the protocol specification, with $\mathcal{S}$ emulating the honest servers and $\mathcal{A}$ emulating the corrupt servers.
14. $\mathcal{S}$ defines G, $\{z_1, \ldots, z_{n-2t}\}$, and $\lambda_{b,m}$ as in step 12 of Convert_Sharings and computes $V_b^{(k)}(i) = \sum_{m=1}^{n-2t} \lambda_{b,m} U_{z_m}^{(k)}(i)$ for each corrupt S_i.
15. Upon receiving $(\texttt{Shares?}, \ell, \ell', K)$ from $\mathcal{F}_{\mathsf{CS}}$, $\mathcal{S}$ replies with shares $V_b^{(k)}(i)$ for each corrupt S_i and for each $(b,k) \in [\ell] \times [K]$.
16. $\mathcal{A}$ provides an output for the environment, which $\mathcal{S}$ forwards to $\mathcal{Z}$.

Theorem 1. *The protocol* Convert_Sharings *UC-emulates the ideal functionality* $\mathcal{F}_{\mathsf{CS}}$.

Proof. We need to show that for any polytime $\mathcal{A}$, there exists a polytime $\mathcal{S}$ such that no polytime $\mathcal{Z}$ can distinguish between interacting with $\mathcal{A}$ (in the execution of Convert_Sharings in the $(\mathcal{F}_{\mathsf{Random}}, \mathcal{F}_{\mathsf{Random}_B}, \mathcal{F}_{\mathsf{Share}_B}, \mathcal{F}_{\mathsf{Open}})$-hybrid model) or interacting with $\mathcal{S}$ (in the ideal world protocol for $\mathcal{F}_{\mathsf{CS}}$).

Output of Honest Servers: We first show that $\mathcal{Z}$ cannot distinguish (except with negligible probability) between the outputs of the honest servers in the hybrid versus the ideal execution. In the ideal execution, the honest servers will output shares of $V_b^{(k)}$ which share the same values as the input polynomials $H_a^{(k)}$, and the $V_b^{(k)}$ will be uniformly random subject to the constraint that they agree with the shares provided by the adversary. Assuming the polynomials $\{U_{z_i}^{(k)}\}_{i\in[n-2t]}$ in step 12 of Convert_Sharings are "correct" in that $U_{z_m}^{(k)}(-a) = H_a^{(k)}(z_m)$, then the $V_b^{(k)}$ will hold the same secrets as the $H_a^{(k)}$ (as shown in the paragraph following the specification of Convert_Sharings). Furthermore, the shares of the corrupt servers are chosen by the simulator to be the same as the shares in the hybrid execution. To see that the $V_b^{(k)}$ are uniformly random given the secrets they hold and the shares of the corrupt players, note that $\mathcal{F}_{\mathsf{Share}_B}$ chooses the $U_i^{(k)}$ for honest S_i uniformly at random given the secrets and corrupt shares. Since there are at least $|G| - t = (n-2t) - t = n - 3t \geq \ell$ honest servers in G, the uniform randomness then follows from the fact that interpolation with coefficients $\lambda_{b,m}$ can be view as multiplication by a hyper-invertible matrix (see the security proof for RandomPairs in [22]).

It remains to show that the polynomials $\{U_{z_i}\}_{i\in[n-2t]}$ are "correct." We show that, except with negligible probability, any dealer S_i who dealt a $U_i^{(k)}$ such that $U_i^{(k)}(-a) \neq H_a^{(k)}(i)$ for some a and k will be eliminated as corrupt in step 10 of Convert_Sharings. The random values $r^{(k)}$ opened in step 5 of Convert_Sharings is a vector in $\mathbb{Z}_p^K$. For some fixed i, let V denote the subspace of $\mathbb{Z}_p^K$ such that

$$\sum_{k=1}^K r^{(k)} U_i^{(k)}(-a) = \sum_{k=1}^K r^{(k)} H_a^{(k)}(i)$$

for each $a \in [\ell']$. Let $\boldsymbol{v} = (v^{(1)}, \ldots, v^{(K)})$ be some vector such that

$$\begin{aligned}\sum_{k=1}^K v^{(k)} U_i^{(k)}(-a) + U_i^{(K+1)}(-a) \\ = \sum_{k=1}^K v^{(k)} H_a^{(k)}(i) + H_a^{(K+1)}(i).\end{aligned}$$

Then the set of all possible vectors $(r^{(1)}, \ldots, r^{(K)})$ in $\mathbb{Z}_p^K$ such the check in step 10 will pass is $\boldsymbol{v} + V$. Now if $U_i^{(k)}(-a) \neq H_a^{(k)}(i)$ for some a and k, then V will be a proper subspace of $\mathbb{Z}_p^K$. Then the probability that a random vector $(r^{(1)}, \ldots, r^{(K)})$ will be in $\boldsymbol{v} + V$ is negligible.

Output of Adversary: The next step is to show that $\mathcal{Z}$ cannot distinguish between the output of $\mathcal{A}$ in the hybrid versus the ideal execution. Since the output of $\mathcal{A}$ is forwarded by $\mathcal{S}$ to $\mathcal{Z}$ in the last step of simulation, it suffices to show that $\mathcal{A}$ cannot distinguish between these two scenarios. We therefore demonstrate that the messages sent from $\mathcal{S}$ to $\mathcal{A}$ have the same distribution as those that would be sent in the hybrid execution.

In step 8 of simulation, $\mathcal{S}$ sends uniformly random values, which is what $\mathcal{F}_{\mathsf{Open}}$ does in the hybrid execution. In step 9, $\mathcal{A}$ corrupts servers, and $\mathcal{S}$ simply forwards their shares to $\mathcal{A}$, which is what $\mathcal{A}$ would see in the hybrid execution. We deal with step 11 below. In step 13, $\mathcal{S}$ follows the protocol specification to emulate the messages that would be sent in the hybrid execution.

In step 11, $\mathcal{S}$ emulates honest servers sending shares of $U_i^{(k)}$ and $H_a^{(k)}$, and $\mathcal{A}$ may adaptively corrupt servers during this process. The shares $H_a^{(k)}(q)$ for $k \in [K]$ that are forwarded from $\mathcal{F}_{\mathsf{CS}}$ are the same as in the hybrid execution. The $H_a^{(K+1)}$ are uniformly random given the corrupt servers' shares in the hybrid execution, and since the $\widetilde{H}_a$ chosen by $\mathcal{S}$ are uniformly random given the corrupt servers' shares, so is $H_a^{(K+1)}$ in the ideal execution, since it is defined by $H_a^{(K+1)} = \sum_{k=1}^K r^{(k)} H_a^{(k)} - \widetilde{H}_a$. Using a similar argument, the $U_q^{(k)}$ for $k \in [K+1]$ are uniformly random given the corrupt servers' shares subject to $U_q^{(k)}(-a) = H_a^{(k)}(q)$ in both the real and ideal worlds. $\square$

B. PDC SECURITY PROOF SKETCH

Description of $\mathcal{F}_{\mathsf{PDC}}$.

The parties that provide input to $\mathcal{F}_{\mathsf{PDC}}$ are the IVS, V, the Ledger Servers, $S_1, \ldots, S_n$, and the users $U_1, \ldots, U_J$, where $J = \mathrm{poly}(\kappa)$.

Ledger Initialization:

1. Receive one copy of the balance ledger from each honest Ledger Server. Each balance ledger entry is of the form (ξ, ζ, ID, b, c), where ξ and ζ "represent" the private and public keys (respectively), ID is the identity associated with the address, b is the current balance, and c is the transaction counter.
2. Initialize the transaction ledger to be empty. The transaction ledger will be of the same format as in the real-world protocol.
3. Receive from V a list L of identities. $\mathcal{F}_{\mathsf{PDC}}$ may receive lists from V at any time during the execution of the ideal process. Upon receiving another list L' from V, update $L \leftarrow L \cup L'$. All lists received from V are forwarded to $\mathcal{A}$.
4. Initialize a set $P := \emptyset$, which will contain ordered pairs of the form (ID, δ), where ID is an identity and δ "represents" a signature on ID.
5. Set the block number: $BlockNum := 1$.

Identity Verification: The following occurs upon receipt of a message of the form $(\texttt{verify\_identity}, ID)$ from a user U.

1. If $ID \notin L$, do nothing (skip the following steps).
2. If $(ID, \cdot) \notin P$, send ID to $\mathcal{A}$. The adversary returns δ, and $\mathcal{F}_{\mathsf{PDC}}$ updates $P \leftarrow P \cup \{(ID, \delta)\}$.
3. For $(ID, \delta) \in P$, send (ID, δ) to U and V.

Interaction with Users: The following messages may be received from users at any time during the ideal execution. Messages will be deleted as in step 3 of Ledger_Server_Loop. The subset $I \subset [n]$ represents the set of servers that are to receive their messages in the current block, while the servers in $\overline{I}$ receive their messages in the next block.

1. Messages of the form $(\texttt{init\_addr}, \mathit{nonce}, V, ID, \delta, I)$. Store this message, and if there is any corrupt S_i such that $i \in I$, then send $(\texttt{init\_addr}, \mathit{nonce}, V)$ to $\mathcal{A}$.
2. Messages of the form $(\texttt{tx}, \mathit{nonce}, \xi_1, \zeta_1, \zeta_2, j, v, I)$. If there is no balance ledger entry with ξ_1 as the private key and ζ_1 as the public key, or if there is no balance ledger entry with ζ_2 as the public key, then delete this message. Otherwise, store this message, and if there is any corrupt S_i such that $i \in I$, then send $(\texttt{tx}, \mathit{nonce}, \zeta_1, \zeta_2, j)$ to $\mathcal{A}$.
3. Messages of the form $(\texttt{check\_bal}, \mathit{nonce}, \xi, \zeta, j_1, j_2, B_1, B_2, I)$. If there is no balance ledger entry with ξ as the private key and ζ as the public key, then delete this message. Otherwise, store this message, and if there is any corrupt S_i such that $i \in I$, then send $(\texttt{check\_bal}, \mathit{nonce}, \zeta, j_1, j_2, B_1, B_2)$ to $\mathcal{A}$.
4. The adversary may send messages on behalf of the corrupt users at any time. The messages may be of any one of the above three forms, with the exception that ID, δ, and v (values that are secret-shared in the real protocol) will be replaced with $\perp$. (This models the fact that $\mathcal{A}$ may deal inconsistent sharings.) $\mathcal{F}_{\mathsf{PDC}}$ deletes or stores these messages according the the same criteria used for messages from the honest users.

Processing Blocks: Upon receiving a message of the form $(\texttt{update block}, v)$ from all honest servers, if $v = \mathit{BlockNum} + 1$, then the following steps are performed:

1. $\mathcal{F}_{\mathsf{PDC}}$ sends messages to $\mathcal{A}$ of the form $(\texttt{init\_addr}, \mathit{nonce}, V, i)$, $(\texttt{tx}, \mathit{nonce}, \zeta_1, \zeta_2, j, i)$, or $(\texttt{check\_bal}, \mathit{nonce}, \zeta, j_1, j_2, B_1, B_2, i)$, which simulates broadcasting those messages for the honest servers. Messages received from users in block $\mathit{BlockNum}$ are broadcast for each honest server with index in I, and messages received in block $\mathit{BlockNum} - 1$ are broadcast for each honest server with index in $\overline{I}$.
2. The adversary may send messages to $\mathcal{F}_{\mathsf{PDC}}$ of the same form as in the previous step, which simulates broadcasting those messages for corrupt server S_i.
3. For each broadcast message satisfying the conditions of step 5 of Ledger_Server_Loop, $\mathcal{F}_{\mathsf{PDC}}$ performs the corresponding action (i.e., adding an entry to the balance ledger for an `init_addr` message, transferring funds for a `tx` message, and returning information to the user for a `check_bal` message). Before doing so, for any message that involved a sharing with shares set to $\perp$ by $\mathcal{A}$, $\mathcal{F}_{\mathsf{PDC}}$ asks $\mathcal{A}$ for a value to replace this with (although $\mathcal{A}$ may still respond with $\perp$, which represents to a corrupt sharing). For an `init_addr` message, $\mathcal{F}_{\mathsf{PDC}}$ chooses random ξ and ζ to output as private and public keys for the user.
4. If a proactive refresh is to occur at the end of this block, $\mathcal{A}$ may choose to decorrupt parties at this step.
5. For each message from an honest user in block $\mathit{BlockNum}$, notify $\mathcal{A}$ as in *Interaction with Users* above if there is any corrupt S_i such that $i \in \overline{I}$.
6. Update $\mathit{BlockNum} \leftarrow \mathit{BlockNum} + 1$.

Corruption: The adversary may choose to corrupt parties at any time during the ideal execution. Upon receiving $(\texttt{corrupt}, S_i)$ from $\mathcal{A}$, $\mathcal{F}_{\mathsf{PDC}}$ sends $(\texttt{corrupt})$ to S_i and sends to $\mathcal{A}$ a copy of the balance and transaction ledgers. In addition, $\mathcal{F}_{\mathsf{PDC}}$ sends $\mathcal{A}$ `init_addr`, `tx`, and `check_bal` messages as in *Interaction with Users* above for any message received from a user in the current block with $i \in I$, and for any message received in the previous block with $i \in \overline{I}$.

We sketch below the simulator for the PDC scheme:

Ledger Initialization: To initialize the corrupt servers' ledger entries, $\mathcal{S}$ provides to $\mathcal{A}$ balance ledger entries like those in $\mathcal{F}_{\mathsf{PDC}}$, except that randomly chosen values representing the corrupt servers' shares are also included. All lists L of identities received are forwarded to $\mathcal{A}$.

Identity Verification: After receiving ID from $\mathcal{F}_{\mathsf{PDC}}$, $\mathcal{S}$ will interact with $\mathcal{A}$ emulating $\mathcal{F}_{\mathsf{Sig}}$ as in [15]. This will result in $\mathcal{A}$ choosing a signature, δ, which $\mathcal{S}$ sends to $\mathcal{F}_{\mathsf{PDC}}$.

Interaction with Users: Upon receiving an `init_addr`, `tx`, or `check_bal` message originating from an honest user, $\mathcal{S}$ relays this message to $\mathcal{A}$, but includes a randomly generated share for each sharing. For each such message originating from a corrupt player, $\mathcal{S}$ forwards a message of the appropriate format to $\mathcal{F}_{\mathsf{PDC}}$.

Processing Blocks: Messages broadcast by honest servers are forwarded from $\mathcal{F}_{\mathsf{PDC}}$ to $\mathcal{A}$, and messages broadcast by corrupt servers are forwarded from $\mathcal{A}$ to $\mathcal{F}_{\mathsf{PDC}}$. For each sub-functionality to invoke, $\mathcal{S}$ receives $\mathcal{A}$'s shares. Based on shares received thus far, $\mathcal{S}$ informs $\mathcal{F}_{\mathsf{PDC}}$ of the value to be shared for each sharing that was initially shared with $\perp$. If the sub-functionality does not terminate, than the failure type is forwarded from $\mathcal{F}_{\mathsf{PDC}}$ to $\mathcal{A}$ (i.e., a sharing was invalid, a user was overdrawing an account, etc.).

Corruption: Upon corruption, $\mathcal{S}$ randomly generates a new share for that server for each ledger entry and each stored user message that has an associated sharing.

The indistinguishability of the view of the environment can be argued as follows: The users who follow the protocol specification will generate uniformly random sharings, and the shares provided to $\mathcal{A}$ by $\mathcal{S}$ are randomly chosen. Since shares provided by $\mathcal{A}$ are replaced with $\perp$ until enough shares are received to determine the value being shared (or whether the sharing is inconsistent), $\mathcal{S}$ can correctly provide input for $\mathcal{F}_{\mathsf{PDC}}$, and hence the correct output will be provided to the environment.