ZeroLedge: Proving Solvency with Privacy

Jack Doerner
University of Virginia
jackdoerner@virginia.edu

Abhi Shelat
University of Virginia
abhi@virginia.edu

David Evans
University of Virginia
evans@virginia.edu

***Abstract*—This paper presents a zero-knowledge proof system by which a bank or exchange can prove properties about its liabilities without leaking any information about individual accounts or more details than necessary about its business. Homomorphism is used to demonstrate certain properties of public commitments to all account balances and to their sum, and verification is distributed among account holders. This paper describes an implementation of the scheme and reports on experiments that demonstrate its practicality up to millions of accounts.**

I. Introduction

In February 2014, the bitcoin exchange Mt. Gox suspended operations and filed for bankruptcy protection. The assets it held were lost and, at the time of writing, have not been fully recovered. The clients of Mt. Gox had previously lost confidence that it controlled enough bitcoin to cover its liabilities in 2011, and in response the exchange had performed an *ad hoc* proof-of-solvency by moving a large quantity of bitcoin in the blockchain [3] (note that this can show control of assets, but does nothing to demonstrate a limit on liabilities). Regardless of whether such evidence of solvency may be considered valid, no similar action was or could be taken on the second occasion.

Bitcoin exchanges lack the regulatory mechanisms available to traditional financial institutions for reinforcing customer trust. Although bitcoin as a system is designed to provide cryptographic assurances, those are lost to any particular coin holder once the coin is transferred to another party, as is the case with many exchanges. Where bitcoin encourages its users to trust in numbers, external regulation necessitates trust in people — particularly in governments or corporations.

The straightforward method for a financial organization to garner trust is *transparency*. Full transparency would reveal all financial information for public audits, but banks may not want details of their operations visible to competitors and customers, and clients may not want their balances and transactions publicly disclosed. A more practical compromise has been to secure a "seal of accountability" from a third-party auditor. However, relying on third-parties to verify claims may not be possible or advisable. For example, the notable accounting firm Arthur Andersen colossally failed its duties when auditing firms such as Sunbeam Corp., Waste Management Inc., and Enron [10].

Contributions. This work explores how an organization can engender trust by *proving* properties of its operations without compromising private information or relying on any third party regulator or auditor. Our goal is to mitigate the problem of implicit trust needed by an institution to prove its own solvency, and the accuracy and completeness of its ledger. The main contribution is to show that zero-knowledge proofs can be designed to efficiently replace ad-hoc proofs for this case. We adopt previously known cryptographic techniques, but this is the first work to demonstrate that zero-knowledge proofs can be used to solve a practical problem at scale.

A proof system as described here cannot ensure that an asset holder is incapable of absconding with funds, nor can it protect against theft of assets by another party, though it does provide a means to detect such a theft. On its own, it does not provide a means to verify a public complaint against a bank or exchange which fraudulently misreports a depositor's balance within the proof itself. The intention is instead to raise the level of accountability for asset holders, and to establish a grounds for trust, where such grounds are not provided by regulation or the economic system itself.

In Sections II and III we show the weaknesses of ad hoc approaches, including the method currently used by bitcoin exchanges. Section IV introduces our design and Sections VI and VII describe our protocol in detail. Section VIII analyzes the cost of our protocol asymptotically and concretely, and Section IX reports on experimental results from our prototype implementation. Our implementation scales to support private liability proofs for a million accounts, costing approximately US$1 in computing resources to produce or verify a proof.

II. Prior Art

We target methods which allow a bank or exchange to prove its solvency without compromising privacy of either account holders or the bank's business. First, we briefly discuss current retail banking regulations in the United States. Then, we present the most advanced method currently used by bitcoin exchanges.

A. Banking Ledgers

The *ledger* of a banking institution records its liabilities in terms of accounts and their respective balances. An institution is *solvent* if the sum of its account balances is less than the sum of its assets. Traditionally, institutions such as banks rely on public trust and a limited form of statutory oversight in order to establish their solvency. With fractional-reserve banking, it is not necessary for a bank to hold assets equal to its liabilities, but sufficient to hold in reserve a fraction of liabilities. For large US banks, the reserve requirement is 10% [7].

In the United States, Reports of Condition and Income are required by statute and collected by the FDIC under the provision of the Federal Deposit Insurance Act [2]. Separate reports

are required and published by the SEC on an annual basis under the provision of Section 13 of the Securities Exchange Act of 1934 [1], including externally audited financial data and statements of financial position.

All financial institutions in the United States are generally accountable to one or more regulatory bodies with power of enforcement. The public must trust that a bank has reported its liabilities in full, and that any external audit has been thorough and correct. However, it is not necessarily possible for a third-party to determine if all accounts have been reported accurately. If, for example, the bank simply excludes certain account holders from their ledger or misreports their balances, only those account holders could object.

This problem is compounded in the case of poorly regulated or unregulated financial institutions—particularly those which have arisen to serve the economies of bitcoin and other cryptocurrencies. As these institutions do not deal in what is considered by most governments to be currency, they are often exempt from the laws and outside the jurisdiction of the regulatory bodies which would otherwise be relevant. The short history of cryptocurrency is already plagued by theft and loss of funds, dishonest reporting, and general financial irresponsibility.

B. *Maxwell Protocol*

In response to the collapse of Mt. Gox, the bitcoin community developed methods for proving the solvency of an exchange. The most notable and successful proposal was originally developed by Gregory Maxwell, and is documented by Zak Wilcox [31]. We refer to it as the *Maxwell Protocol*.

The Maxwell Protocol aims to allow Bitcoin exchanges to prove their solvency without compromising the privacy of account holders. It consists of two parts; the first is a proof of liabilities, in which the exchange publicly discloses its liabilities in a way designed to ensure that the exchange cannot falsify them without detection. The second is a proof of assets, in which the exchange proves control of sufficient funds by signing a public message with the private key (or keys) associated with the bitcoin it holds. So long as the liabilities are less than the assets, the exchange is solvent.

In the Maxwell Protocol, the proof of liabilities takes the form of a modified Merkle tree [22] in which each leaf node represents a single account. Figure 1 illustrates an example Merkle tree, as used in the Maxwell Protocol. The hash h_i of the leaf node associated with account i is derived using hash function H from its balance, v_i, account ID, x_i, and a nonce, r_i, all of which are known to the account holder:

$$h_i = H(x_i, v_i, r_i)$$

The standard implementation of H simply concatenates its arguments with a delimiter, and passes the result to SHA-256.

The value of each interior node j is the sum of its children's values, and its hash is derived from its own value and the hashes of its children:

$$v_j = v_{\texttt{left}} + v_{\texttt{right}}$$
$$h_j = H(v_j, h_{\texttt{left}}, h_{\texttt{right}})$$

Thus, the value of the root is the sum of all account balances included in the proof. The hash and value for the root node are published as the exchange's public commitment.

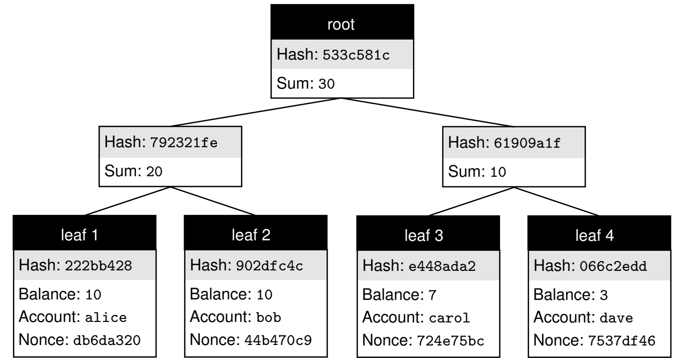


Fig. 1: Merkle Tree used by the Maxwell Protocol

The exchange sends to each account holder the values and hashes for all interior nodes between their own leaf and the root, as well as the values and hashes for the siblings of these nodes. With this information, account holders can verify that their balance has been included in the total, and that none of the siblings' values are negative or large enough to cause an integer overflow. If a significant fraction of account holders check regularly, then they may be relatively certain the exchange has not reported its liabilities fraudulently.

Although the Maxwell Protocol has been implemented [6, 20, 30] and put into use by several bitcoin exchanges [8, 13], it has several serious problems, which we discuss in Section III.

C. *TPM Approach*

Decker et al. [15] suggest a variant of the Maxwell Protocol which uses a trusted platform module (TPM) to execute and cryptographically seal the calculations as they occur. Whereas the standard Maxwell Protocol accumulates balances through the tree and publishes the total, they propose to include the individual balances only at the leaves and accumulate the total privately. The TPM is used to ensure that this summation is performed honestly. The TPM is also used to privately verify the assets held by the exchange, and to compare them to the liability total. The system outputs true if the exchange is solvent, and false if it is not. This sealed output is published along with the Merkle tree. Account holders must verify that their account is included, that the calculation reports solvency, that the executed code is valid, and that the output is signed by a trusted TPM.

This approach attempts to ameliorate several of the concerns which we will present in Section III. Unlike the standard Maxwell protocol, it leaks little or no financial information belonging to either the exchange or the account holders. While it does prevent the exchange from including negative balances, the exchange may still cheat by excluding accounts belonging to clients it believes will not check (this problem, however, is unavoidable in any scheme). On the other hand, this method requires the account holders to trust a black box hardware module that is physically controlled by the exchange. It also

mandates additional, direct interaction between the prover and the verifier to establish the trustworthiness of the TPM.

III. Problems with the Maxwell Protocol

The Maxwell Protocol suffers from several serious issues, both in its general design and in how it is currently instantiated. We discuss general architectural weaknesses in Section III-A and present a concrete attack in Section III-B.

A. Architectural Weaknesses

The design of the Maxwell Protocol inherently leaks some information about user accounts and the exchange; it also provides less protection than desired against an adversarial exchange. Several of the weaknesses stem from the design providing the prover (exchange) with the opportunity to make arbitrary decisions in structuring the proof, without revealing those decisions to the verifier.

Account Information Leaks. Because the proof is structured as a Merkle tree, each verifier learns the balance belonging to their sibling leaf. For this reason, the accounts must be randomly shuffled for each new publication of the proof. Even with shuffling, it is likely that verifiers can discover something about the distribution of funds within the exchange, especially in collusion with one another. Jesse Powell, the CEO of the bitcoin exchange Kraken, has explicitly cited account holder privacy as the primary reason that his company has not implemented the Maxwell Protocol [25].

Information leakage can be mitigated by splitting each account's balance among multiple leaf nodes, and requiring verifiers to check each of their leaves individually. However, this balance-splitting modification is not part of the standard formulation, nor is it actually used in practice. In addition, this modification makes another layer of arbitrary decisions available to the prover, and increases the power of the attack described in Section III-B

Exchange Information Leaks. Both the total assets and the total liabilities of the exchange must be revealed. This requirement is not unlike those imposed on traditional banks and exchanges by regulation. Nevertheless, it is a requirement most bitcoin exchanges would like to avoid, especially if proofs are published frequently.

Dependence on Complete Account-Holder Verification. The Merkle-tree approach requires that responsibility for verifying both the integrity and the correctness of the proof be distributed to all account holders. Without universal participation, neither can be achieved. In the case of correctness, this is necessary; a distributed proof of correctness is precisely what the scheme aims to achieve. Distributing the verification of integrity, however, opens the door for attacks, such as the one we introduce in Section III-B.

Interactive Verification. Each account holder receives an individual proof from the exchange containing only the nodes between their own leaf node and the root. In practice, this means that each account holder must *request* their individual proof, even if implicitly, and thereby reveal to the exchange that they intend to verify the proof. A malicious exchange could take advantage of this information by, for example, omitting the accounts of those it believes are unlikely to verify the proof, returning network errors when it receives a request from an account holder who might discover its fraud, or by ensuring that known verifiers' accounts are reported honestly while others are not.

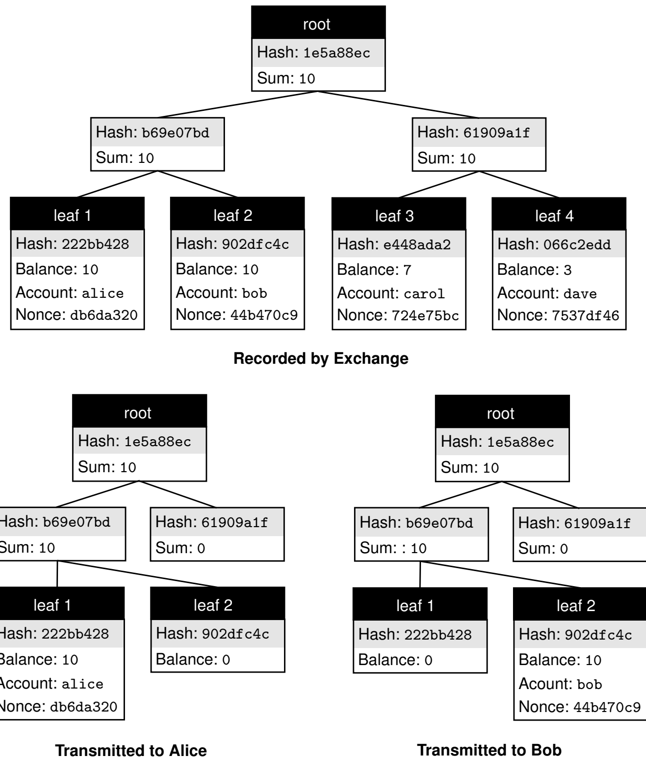


Fig. 2: Hiding Liabilities in the Maxwell Protocol

B. An Attack on the Standard Formulation

There is a flaw in the Maxwell Protocol as described and implemented that enables an exchange to misrepresent its liabilities. The trouble arises from the function used to generate interior node hashes:

$$H(v_j, h_{\texttt{left}}, h_{\texttt{right}})$$

This function effectively ensures the inclusion in node j of the two branches below it, and of the sum v_j. However, it fails to establish the ownership of the branches' individual values. That is, for interior nodes j and k and their child nodes a, b and c, d, respectively, such that the value of a is equal to c and b to d, but the hashes of the second pair are reversed, both the values and hashes of j and k will be identical, in spite of the fact that the child nodes of j and k differ. Therefore, multiple trees can be constructed with the same root hash and value, as illustrated in Figure 2.

Implications. This weakness can be used to hide values within the tree, proving their inclusion while excluding them from the sum. For any two subtrees of equal value, the prover can falsely

reduce or eliminate the value of one subtree. The prover then sends to each verifier a proof containing the original (correct) value for their own subtree, and the falsified value for the sibling subtree. All verifiers will still arrive at the same root hash, and none of them will be able to independently detect the falsification of sibling nodes.

The simplest attack is to combine identical balances opportunistically. Given that an established exchange will have a large number of accounts, and that people tend to prefer whole numbers, it is not inconceivable that a significant sum could be hidden among the leaves alone, especially if the exchange is capable of manipulating balances by way of fees, suggested transaction amounts, or other social or policy mechanisms.

The true manipulative power of the exchange arises when the weakness is exploited repeatedly at multiple levels, compounding the value hidden. It can be used to reduce the published value of a subtree with equal branches by as much as half. Therefore, if a subtree with equal branches has a sibling subtree with a value less than its own but greater than half of its own, the two subtrees can be made equal and their composite value can be fraudulently reduced in turn.

The power of this attack dramatically increases when fake nodes are introduced. For any two unequally-valued subtrees, it is possible to replace a single leaf node in the lesser of the two with a fake account which makes up the difference, thereby fulfilling the condition for hiding funds when they are combined. By using false accounts and exploiting the vulnerability at multiple levels, it is possible to hide an arbitrary sum, limited only by the size of the largest single account balance.

The balance-splitting modification of the Maxwell Protocol grants a fraudulent exchange even greater power to hide funds, permitting it to split accounts in such a way that they form equally valued subtrees where they would not otherwise do so. Thus, an exchange can hide an arbitrary sum without artificially inflating the size of the tree, as is required by the use of fake accounts.

We implemented a fraudulent proof generator as a proof of concept, and confirmed that the fradulent proofs it generates are declared valid by a public implementation [20] of the standard Maxwell Protocol verifier. Given a list of accounts and balances, our fradulent proof generator is able to construct a "maximally hiding" proof, in which the exchange commits to a total liability exactly equal to that incurred by the single largest account, while fraudulently proving that all of the other accounts have been included as well.

Simple Fix. The problem of ownership ambiguity can be solved with an alteration to the standard formulation of the Maxwell Protocol. Modifying the interior node hash function to include subtree values separately effectively ties them to their respective subtrees. This prevents the attack since

$$H(v_{\texttt{left}}, h_{\texttt{left}}, v_{\texttt{right}}, h_{\texttt{right}}) \neq H(v_{\texttt{right}}, h_{\texttt{left}}, v_{\texttt{left}}, h_{\texttt{right}})$$

However, this construction does not mitigate the other concerns presented in Section III-A.

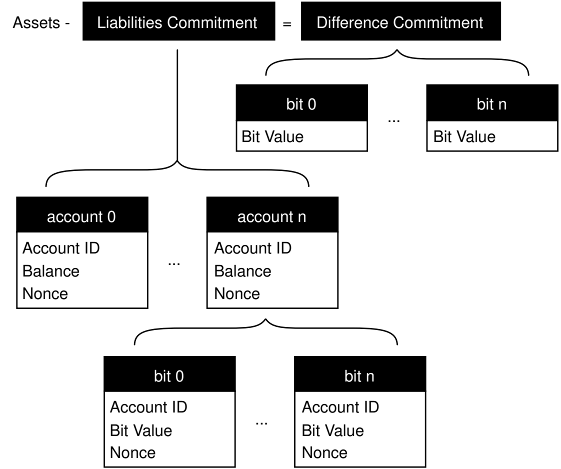


Fig. 3: ZeroLedge Proof Structure

IV. Overview

ZeroLedge is a non-interactive, zero-knowledge proof of solvency that addresses the deficiencies in the Maxwell Protocol. While maintaining the properties of public commitment and distributed verification, it abandons the Merkle tree structure and the transmission of individual proofs in favor of a single, monolithic, public proof document in which balances are committed privately and manipulated homomorphically. We focus on the proof of liability, and support proofs that establish only that the total liabilities are less than a given reserve input value (so, unlike the Maxwell Protocol, our protocol does not leak the actual total liabilities). When employed by bitcoin exchanges, our protocol could include the same proof of assets used by the Maxwell Protocol to produce the reserve input value.[1]

Our goal is to provide institutions with a way to privately prove that their liabilities do not exceed a required threshold. Each depositor knows the liability resulting from their own account balance, and can verify that this liability has been included in the report. This must be done without revealing to any depositor any information about another depositor's account.

In our system, illustrated in Figure 3, the prover generates commitments to identifier/value pairs for every account, and to the sum of all accounts (the total liabilities). The prover also generates commitments to the bitwise expansions of each account balance and the bitwise expansion of the difference between the assets and the total liabilities. These commitments are compiled along with zero-knowledge proofs of their validity into a single *proof transcript*, which is published in a public place where it cannot be altered or retracted.

Each account holder may use this proof transcript along with an opening, provided by the institution, to verify that their

[1] A private summation of assets such as the one suggested by Decker et al. [15] can also be combined with ZeroLedge, assuming that the summation of assets is trustworthy. Under such an arrangement ZeroLedge will reveal no financial information of any kind.

own account commitment exists and corresponds to the correct value, and that this value is included in the sum commitment. In addition, any third party can verify that every account has a nonnegative balance, and that no uncommitted balances have been included in the sum. Finally, a third party can verify that the overall sum matches or is less than a particular published value, corresponding the the assets of the institution.

This proof does not in itself provide recourse in the case that an account is omitted or a balance is incorrect. Instead, the institution must provide some secondary evidence to the account holder of the account status and balance, perhaps in the form of statements and transaction history signed by both parties. Regardless, the opening to each commitment should be signed by the institution on delivery to ensure each account holder has a verifiable record which can be divulged to other parties in the case that this proof fails.

V. Preliminaries

The main guarantees of our protocol arise directly from our use of cryptographic commitments to account balances and zero-knowledge proofs of their correctness. In this section, we define a ledger and briefly review security properties of commitments and zero-knowledge proofs.

A. Ledgers

A *ledger* records financial transactions between parties. For our purpose, we restrict the definition of a ledger to only record financial balances owed to multiple individual depositors by a single asset holder:

Definition 1 (Ledger): A ledger $\mathscr{L}$ consists of a sequence of pairs $\mathscr{L} = ((x_1, v_1), \ldots, (x_n, v_n))$ where i is an index, x_i is an account identifier for account i, and v_i is an integer corresponding to the value of the account.

The string x_i can be viewed as an account number or name and v_i is its balance. We model both as private strings known by P and the account holder. Thus i can be considered as the public handle to the account which we use interchangeably with the account owner of x_i. This allows participants in the system to refer to account i without knowing its account number.

A ledger is sound if all account values are non-negative:

Definition 2 (Sound Ledger): A sound ledger is a ledger $\mathscr{L} = ((x_1, v_1), \ldots, (x_n, v_n))$ for which $v_i \geq 0$ for $i \in [1, n]$.

Thus, the institution knows its entire ledger and account holder i knows (x_i, v_i). Our notion of a sound ledger models how a bank and its customers share information about deposits.

B. Commitment Schemes

In a commitment scheme, a *sender* can place a secret message s into a commitment by running a randomized algorithm $c \leftarrow \mathsf{Com}(s; r)$ where r represents the randomness used to form the commitment, and then sending c to a *receiver*. Later, the sender can *open* the commitment and reveal the message s to the receiver by sending r.

A commitment scheme possesses two security properties: *hiding* and *binding*. The first property guarantees that the receiver cannot derive any information about s from c. The second property guarantees that for a given commitment c, it is not possible for an efficient sender to find a commitment c^* and two pairs, (s_1, r_1) and (s_2, r_2) such that r_i opens c^* to value s_i and $s_1 \neq s_2$. The commitment schemes we use require a common parameter which is generated by an algorithm Gen and we assume its output is implicitly given to Com.

Definition 3 (Commitment Scheme): We say that a tuple of probabilistic polynomial time algorithms $(\mathsf{Gen}, \mathsf{Com})$ is a *non-interactive commitment scheme* if:

1) Computational Binding: For every efficient adversary A there exists a negligible function μ such that for every $n \in \mathbb{N}$,

$$\Pr\left[\begin{array}{c} p \leftarrow \mathsf{Gen}(1^n); (s_0, r_0), (s_1, r_1) \leftarrow A(1^n, p): \\ s_0 \neq s_1 \wedge \mathsf{Com}(s_0; r_0) = \mathsf{Com}(s_1; r_1) \end{array}\right] \leq \mu(n)$$

2) Computational Hiding: For any pair of messages $s_0, s_1 \in \{0,1\}^*$, the following ensembles are computationally indistinguishable.

$$\{p \leftarrow \mathsf{Gen}(1^n) : p, \mathsf{Com}(s_0)\}_{n \in \mathbb{N}}$$
$$\{p \leftarrow \mathsf{Gen}(1^n) : p, \mathsf{Com}(s_1)\}_{n \in \mathbb{N}}$$

Our scheme uses Pedersen commitments [24], which have been proven to have both required properties:

Theorem 1: Assuming the hardness of the discrete logarithm problem, the Pedersen commitment is a perfectly-hiding and computationally-binding commitment scheme (Proven by Pedersen [24]).

C. Non-Interactive Zero Knowledge Protocols

Zero-knowledge protocols allow a prover P with a witness w for a language L to convince a verifier V that $x \in L$ without revealing any other information. Such proof systems satisfy three conditions (specified formally in Definition 4): *completeness*, *soundness*, and *zero-knowledge*. Informally, these properties ensure that true statements can be proven, malicious provers cannot produce proofs of false statements, and that malicious verifiers cannot learn anything more than the fact that a theorem statement is true. Additionally, in a non-interactive protocol, the prover sends only one message to the verifier. In our case, we even allow a malicious verifier to choose the theorem statements adaptively.

In the Random Oracle model, all parties have access to an oracle $\mathbb{O}_{p,p'} = \{\mathbb{O}_n\}_{n \in \mathbb{N}}$ where $\mathbb{O}_n$ is simply uniform distribution over functions $\{0,1\}^{p(n)} \to \{0,1\}^{p'(n)}$ (where p and p' are polynomials).

Definition 4 (NIZK in the Random Oracle Model): A tuple $(P, V, \mathbb{O}, S)$, is called a non-interactive zero-knowledge proof system (NIZK) in the random oracle model for a language L with witness-relation $R_L(\cdot)$ if the algorithm V is deterministic polynomial-time, and P is probabilistic polynomial-time, and $\mathbb{O} = \{\mathbb{O}_n\}_{n \in \mathbb{N}}$ is a sequence of distributions $\mathbb{O}_n$ over oracles $\{0,1\}^{p(n)} \to \{0,1\}^{p'(n)}$ (where p and p' are polynomials), such that the following holds:

- *Completeness*: There exists a negligible function μ such that for every $n \in \mathbb{N}$, $x \in L$ s.t. $|x| = n$, every $w \in R_L(x)$ and every $\mathsf{tag} \in \{0,1\}^n$,

$$\Pr[\begin{array}{l}\rho \leftarrow \mathbb{O}_n;\ \pi \leftarrow P^\rho(\mathsf{tag}, x, w)\ : \\ V^\rho(\mathsf{tag}, x, \pi) = 1\end{array}] \geq 1 - \mu(n)$$

- *Soundness*: For every algorithm B, there exists a negligible function μ such for every $n \in \mathbb{N}$, $x \notin L$ s.t. $|x| = n$ and every $\mathsf{tag} \in \{0,1\}^n$,

$$\Pr[\begin{array}{l}\rho \leftarrow \mathbb{O}_n;\ \pi' \leftarrow B^\rho(\mathsf{tag}, x)\ : \\ V^\rho(\mathsf{tag}, x, \pi') = 1\end{array}] \leq \mu(n)$$

- *Simulatability*: there exists a probabilistic polynomial-time (p.p.t.) simulator S such that the following two ensembles are computationally indistinguishable by polynomial-sized circuits that have oracle access to ρ:

$$\{\rho \leftarrow \mathbb{O}_n;\ \pi \leftarrow P^\rho(\mathsf{tag}, x, w, z)\ :\ (x, \pi, \rho)\}_{n \in \mathbb{N}, x \in L, z \in \{0,1\}^*}$$
$$\{\pi', \rho' \leftarrow S(x)\ :\ (x, \pi', \rho')\}_{n \in \mathbb{N}, x \in L, z \in \{0,1\}^*}$$

All of the protocols we describe in this paper are modifications of Schnorr proofs [26] and will therefore satisfy the basic, 3-round, special-sound, honest-verifier notion of zero-knowledge protocol known as a *Sigma-protocol* (for a complete definition, see Cramer et al. [14]).

We compile such Sigma-protocols into non-interactive zero-knowledge proofs in the random-oracle model (Definition 4) using the Fiat-Shamir technique [16]. This technique can be applied to any public-coin protocol by defining the verifier's random messages to be chosen by applying a hash function to the previous messages of the protocol. In the case of three-round protocols, the hash function is applied to the theorem statement and the first message in order to pick the challenge message.

VI. Protocol

In this section we describe a high-level version of our protocol and then explain how desired properties follow. Section VII provides a specific stepwise formulation.

Notation. We use v_i to denote an integer value representing the balance associated with account i, and $v_{i,j}$ is the jth bit of v_i. We use C_i to denote a public commitment to ledger entry i. $D_{i,j}$ is a public commitment to bit j of ledger entry i. We use C_T to represent public commitment to total liability of the institution and $C_{T,j}$ is the public commitment to bit j of the total liability. Similarly, C_Z represents a public commitment to Z, the difference between the stated assets and total liabilities (i.e., $Z = X - \sum_i v_i$, where X is the stated assets), $v_{Z,j}$ is the jth bit of Z, and $D_{Z,j}$ is the public commitment to bit j of Z. We use ℓ to denote the fixed number of bits used to represent all balance values and sums, as well as the number of bitwise commitments to each ledger entry and to the total liability.

A. Protocol Overview

Our scheme achieves its guarantees through a two-step process. First, the institution (which assumes the role of the prover) *commits* to its ledger,

$$\mathsf{Com}(\mathscr{L}) = C_1, \ldots, C_n$$

and a value X representing the total assets, and computes a non-interactive zero-knowledge proof π that the committed ledger and value X are in the language L_S, of valid, solvent ledgers (specified in Definition 5):

$$(\mathsf{Com}(\mathscr{L}), X) \in L_S$$

Definition 5 (Solvent Ledger): The specific language for which we design a zero-knowledge proof in this paper is the set of all commitments corresponding to sound ledgers whose liability does not exceed the required threshold:

$$L_S = \left\{ (C_1, \ldots, C_n, X) \;\middle|\; \begin{array}{l} \exists((x_1, v_1, r_1) \ldots (x_n, v_n, r_n)) \text{ s.t.} \\ \quad \forall i : v_i \geq 0 \wedge C_i = \mathsf{Com}(x_i, v_i; r_i) \\ \quad \wedge \sum_i v_i \leq X \end{array} \right\}$$

Our proof comprises the intersection of languages L_1, L_2, L_3 and either L_{4E} or L_{4I}, specified in Section VII.

Second, institution provides each account holder i with an opening to the commitment C_i to account balance v_i. This allows account holder i to verify that the ledger that was committed accurately reflects the institution's obligation to i.

We now argue why this general scheme satisfies the properties we desire.[2]

B. Integrity Properties

Here, we expound the functionality goals of our scheme, and argue that when they are met, a malicious prover could not publish fraudulent proofs without high probability of detection.

Distributed Verifiability of Correctness. Any account holder should be able to verify that the balance associated with their own account is included and correct.[3] ZeroLedge achieves this property by providing account holder i with an opening, r_i, to the commitment corresponding to account i: $C_i = \mathsf{Com}(x_i, v_i; r_i)$. Furthermore, the accounts must be represented in the proof in such a way that they are unambiguously identifiable to their owners. ZeroLedge achieves this by including a unique account identifier, x_i, in each commitment.

Verifier anonymity. The scheme should not require that any verifier interact with the prover in such a way that their identity (as an account holder) is revealed. The prover should not be able to determine which account holders verify the proof, unless it has out-of-band information.

ZeroLedge achieves this property because both the commitment protocol and the proof of correctness are non-interactive.

[2] In the full version of this paper, we will provide an ideal functionality that captures these notions. In particular, this involves making our NIZK proof online simulation-extractable to show UC-security. This would require a small modification to our protocol, and is relatively efficient and easy in the random-oracle model. In this paper we provide an informal argument and focus more on the design and experimental analysis of the scheme.

[3] If some account holder i does not bother to participate in verifying that their private account balance is included, then the scheme is not sound. This seems fundamental. If we ensure that balance v_i is private, then only i or some third party with i's private opener can verify the v_i that is implicitly published by the institution. Even if an institution agrees to publish its ledger *in the clear* without any regard for privacy, it is impossible to verify whether the institution has simply omitted accounts or reduced liabilities for holders whom it knows are unlikely to check the proof.

Thus, any account holder can use an anonymous network like Tor to access the proof, or obtain it from a third party. In contrast, because the Maxwell Protocol is an *interactive proof* in which the verifier first discloses their identity, a malicious institution could ensure that the balances of known verifiers are included correctly, while omitting known non-verifiers from the proof.

For the verification of the inclusion of each account, the openings to the commitments must be sent to account holders privately, non-interactively, and unsolicited. Any method which delivers these nonces "by request" allows the prover to determine which users verify its proofs. This requirement can be met by using a unique pseuro-random seed to generate all of the openings for each account. The seed can be determined and shared with the account owner when the account is created, allowing the prover and account holder to generate identical openings with no further communication.

Similarly, each account's index in the proof should be pre-agreed. New accounts can be added to the proof at the end, and accounts which have been closed can remain with null balances or be replaced by dummies until the next index refresh.

Global Consistency. Verifiers must be able to determine unambiguously and without revealing private information to one another that they are reading the same proof. The prover should not be able to construct multiple proofs such that they appear to be the same to any verifier or verifiers. Furthermore, proofs should be constructed in such a way that they cannot be altered after the fact in any way invisible to the verifiers. It is through a failure to achieve global consistency that the attack on the Maxwell Protocol described in Section III-B arises. ZeroLedge achieves global consistency by constructing a single, monolithic, non-interactive proof document and requiring its publication in a public place.

When used by cryptocurrency exchanges, we recommend the blockchain itself as a publication medium for assuring the authenticity, immutability, and universal visibility of proof transcripts. While proofs will certainly be too long to inject into the blockchain in their entirety, it should be possible to inject hashes of the documents at minimal expense. This has the additional property of establishing an approximate time of publication.

Public Verifiability of Integrity. Any person should be able to independently verify that every entry in the ledger has a nonnegative balance, and that the sum of balances is correctly calculated. As a result, verification of the integrity of a proof can be performed by semi-trusted third parties (such as government regulators) if desired, and the participation of account holders is required only to ensure that all accounts have actually been included.

ZeroLedge achieves this by providing a non-interactive zero-knowledge proof that commitments correspond to non-negative values. Any verifier—even one without an account—can check the NIZK proof by running the V^{ρ} method on the proof.

C. Confidentiality Properties

Our protocol is designed not to leak any information about other accounts to participants, and to leak as little information as possible about the institution. The only substantial information leaked by our protocol is the number of accounts, which may be padded, the value of the assets,[4] and whether the institution is solvent.

Private Commitment to Liability Total. The proof should be constructed in such a way that the total liability can be committed privately, thereby minimizing the information revealed by the institution. ZeroLedge achieves this by never committing to total liabilities in such a way that the commitment can be opened. The computational hiding property of our commitment scheme (discussed in Theorem 1) ensures that it is infeasible to open the commitment without prior knowledge of the committed value.

Privacy. No verifier or group of verifiers in collusion should be able to discover any information about any account holder or group of account holders (except perhaps themselves) at any time. In ZeroLedge, all balances and the sum are protected by the zero-knowledge property of the proof system. In contrast, the Merkle tree used by the Maxwell Protocol makes it highly susceptible to collusion attacks.

Trusted Parameters. The system should not involve any global trusted parameters that have secret trapdoors. For example, we do not consider it practical to use an RSA modulus n for which it is assumed that nobody knows the factorization as a common parameter, because it cannot be verified that the prover does not know the factorization secretly. Similarly, we studied but dismissed schemes that require parameters with secret trapdoors to be published globally. A prover with the trapdoor information could cheat on the proof, and a verifier with the secret trapdoor might be able to learn information about the ledger. On the other hand, we believe that it is reasonable to choose an established elliptic curve as a global parameter and to use fundamental mathematical constants as arbitrary values (see Section VII-A).

D. Instantiation Overview

There are various cryptographic approaches to each step as enumerated in Section VI-A. We choose to use the Pedersen commitment scheme [24] in an elliptic curve group, and to use Schnorr proofs and the Fiat-Shamir heuristic to produce non-interactive zero-knowledge proofs for each of the statements.

In order to prove that a commitment C_i corresponds to $v_i \geq 0$, we take a textbook approach and commit to each of the ℓ bits $v_{i,j}$ of v_i. Pedersen commitments are homomorphic: if $c_1 = \mathsf{Com}(m_1, r_1)$ and $c_2 = \mathsf{Com}(m_2, r_2)$, then $c_1 * c_2 = \mathsf{Com}(m_1 + m_2, r_1 + r_2)$. The Pedersen commitment also allows multiplication by scalars, so that if $c_1 = \mathsf{Com}(m_1, r)$, then ${c_1}^2 = \mathsf{Com}(2m_1, 2r)$. This provides a straightforward method to prove that that a commitment to v_i and ℓ commitments to $v_{i,j}$ represent the same value. Finally, in order to prove that

[4] Assuming a public proof-of-assets is employed, as in the Maxwell Protocol

each commitment to $v_{i,j} \in 0,1$, we will use a standard OR-method for composing Schnorr proofs [14]. This permits us to conclude that $v_i \in [0, 2^\ell]$.

Alternative Implementations. The largest contribution to our proof complexity arises from the commitment to each bit and the proof that it is either 0 or 1, as explained in Section VII. There are a number of alternative methods for a prover to convince a verifier that commitment C_i contains a value v_i in the range $v_i \in [0, 2^\ell]$. We now briefly discuss alternative approaches to solving this problem and why they are not well-suited to our scheme.

Under the Strong (or sometimes called flexible) RSA assumption, Boudot [9] provided an efficient proof based on the observation that any positive number can be composed as the sum of four squares. Thus, to show that a secret v_i lies in $[0, B]$, one provides commitments to v_i and $B - v_i$ and to the 8 numbers $s_{(1,1)}, \ldots, s_{(1,4)}$ and $s_{(2,1)}, \ldots, s_{(2,4)}$, the sum of whose squares are equal to v_i and $B - v_i$ respectively. Okamoto and Fujisaki [17] have shown that when the commitments and the proof are performed in a group where the order is not known to the prover (e.g., when the modulus is an RSA modulus), these relations hold over the integers and thus one can really assert that v_i and $B - v_i$ are positive. Groth [18] optimizes this protocol by exploiting the fact that special integers can be written as the sum of 3 squares instead of 4 squares.

The first major drawback to this approach is that the protocol requires an RSA modulus n for which nobody knows the factorization. Thus, the protocol requires a *trusted setup* that cannot be verified externally (one cannot verify that nobody, neither prover, the setup agency, nor some client knows the factorization of n). We want our protocols to be designed such that all system parameter choices can be verified, and none require secret coins to produce.

The second drawback of this protocol is that the Rabin and Shallit algorithm (typically used to find the squares which sum to the secret) takes time $O(k^4)$ where k is the size of the interval. Lipmaa [21] provides an optimization to find the squares, but in practice, these algorithms are too expensive. The third drawback is the size of each element in an RSA group (3000 bits), as compared to the 256 bits needed to represent elements in our elliptic curve group. Thus, even though the RSA approach needs only a few group elements for the proof, the overall size is roughly comparable to our own proof size.

Schoenmakers [27, 28] described several recursive relations that can be used to reduce the number of basic Schnorr proofs required when committing to the individual bits of the secret. In particular, he represents the upper bound B of the positive range $[0, B)$ as either the product or the sum of two numbers. By following this scheme recursively he decreased the required work. However, the overall communication load of his protocol is still $O(k)$, where $2^{k-1} < B \leqslant 2^k$. We note that some of his techniques for reducing certain ranges to other, more convenient ranges can be used with any range proof technique.

The protocols of Camenisch et al. [11] and Chaabouni et al. [12] can be used to construct range-proofs in a prime-order group (such as our own) that are both asymptotically and practically more efficient than the ones we use, but they are *private-coin interactive proofs* in which the verifier must send specially-constructed signatures to the prover. No such entity could provide those elements in our case without requiring *extra trust assumptions* or a trusted setup phase.

It is possible that Groth-Sahai [19] proofs may improve the bandwidth efficiency of our protocol—in particular, that protocol provides a special method for proving that a commitment is either 0 or 1. However, their proofs require common reference strings with group elements selected in a specific manner and they require operations to be performed in a more complicated bilinear pairing group that satisfies the SXDH assumption. The cost of a proof involves (a) a common reference string which includes a vector of group elements in $\mathbb{G}_1$, and $\mathbb{G}_2$, (b) commitments to the values in both groups, and a proof in both groups; in total, the proof requires 6 elements in both $\mathbb{G}_1$ and $\mathbb{G}_2$, with elements in $\mathbb{G}_2$ being twice the size (512 bits) of those in $\mathbb{G}_1$. In contrast, our proof only requires eight elements of a size equivalent to $\mathbb{G}_1$.

Recent work on zk-SNARK systems [4, 23] is also relevant. However, these systems require a "key generator" that samples a correlated proving key and verification key for the circuit being verified. This key generator requires private coins to ensure soundness and thus is not suitable for our problem.

VII. Protocol Specification

In this section we lay out the protocol in detail. It is framed as an interaction between a prover, P, and a verifier, V, who may or may not be the holder of a particular account. However, as the protocol is noninteractive, it must be remembered that the actions of the two parties are asynchronous. The prover's steps result in a single, monolithic proof transcript, and it is upon this transcript that the verifier performs its own actions after the fact.

A. Generation of Constants

Let $\mathbb{G}$ be a group of prime order q and g, h, f be elements for which no one knows the mutual discrete logarithms in $\mathbb{G}$ with respect to any of the bases. Such "nothing-up-my-sleeve" numbers can be derived from the expansion of well-known constants such as $\pi, e, \sqrt{2}$, etc. and included in our cryptographic hardness assumption.[5]

B. Commitments

The prover begins by generating and publishing Pedersen commitments to its ledger. These public commitments computationally bind the prover to a single ledger.

Commitment to Ledger Entries. P commits to the ledger entry (x_i, v_i) by computing x'_i, r'_i, C_i and publishing C_i:

1) Pick nonces $r_{i,j}$ from $\mathbb{Z}_q$,

[5] As a result, our instantiated scheme departs from our theoretical one in that our instantiated scheme is no longer secure against non-uniform adversaries (who receive as advice the particular discrete logarithms).

2) Compute

$$x_i' = x_i(2^\ell - 1)$$
$$r_i' = \sum_{j=0}^{\ell-1}(2^j r_{i,j})$$
$$C_i = g^{x_i'} h^{v_i} f^{r_i'}$$

Commitment to Ledger Entry Bits. P commits to the bits of ledger entry i by computing and publishing the values $D_{i,0}, \ldots, D_{i,\ell-1}$.

$$D_{i,0} = g^{x_i} h^{v_{i,0}} f^{r_{i,0}}$$
$$\ldots$$
$$D_{i,\ell-1} = g^{x_i} h^{v_{i,\ell-1}} f^{r_{i,\ell-1}}$$

Commitment to Difference Bits. P commits to the bits of Z by computing and publishing the values $D_{Z,0}, \ldots, D_{Z,\ell-1}$.

$$D_{Z,0} = g^{-\sum_i x_i} h^{v_{Z,0}} f^{-\sum_i r_{i,0}}$$
$$\ldots$$
$$D_{Z,\ell-1} = g^{-\sum_i x_i} h^{v_{Z,\ell-1}} f^{-\sum_i r_{i,\ell-1}}$$

Cost. Significant computational work may be saved by caching the values of $g^{x_i'} h^1$ and $g^{x_i'} h^0$, which will be consistent for all bits in a given ledger entry. This reduces the required number of exponentiations to one per bit per ledger entry, plus one per ledger entry. The values of $g^{-\sum_i x_i} h^1$ and $g^{-\sum_i x_i} h^0$ can be cached in a similar manner.

C. Proofs of Commitment Validity

Now the prover proves that the values $C_i, D_{i,j}, D_{Z,j}$ for all ledger entries i and bits j that are published in Section VII-B are proper Pedersen commitments and not arbitrarily chosen values. That is, the prover proves knowledge of the identifier, nonce, and balance or bit associated with each commitment.

Ledger Commitments. P proves that each C_i is a commitment to a valid ledger entry with an identifier $x_i' = x_i(2^\ell - 1)$, balance v_i, and nonce r_i'.

$$L_1 = PoK\left\{(x_i', v_i, r_i') : C_i = g^{x_i'} h^{v_i} f^{r_i'}\right\}$$

1) P picks a random $b_1, b_2, b_3 \in \mathbb{Z}_q$, computes γ, and sends γ to V.
$$\gamma = g^{b_1} h^{b_2} f^{b_3}$$
2) P generates a challenge c by using the random-oracle H and the tag 0^n.
$$c = H(g, h, f, C_i, \gamma, 0^n)$$
3) P computes and sends z_1, z_2, z_3 to V.
$$z_1 = b_1 + c x_i'$$
$$z_2 = b_2 + c v_i$$
$$z_3 = b_3 + c r_i'$$
4) V verifies
$$g^{z_1} h^{z_2} f^{z_3} \stackrel{?}{=} C_i{}^c \gamma$$

Ledger Bit Commitments. P proves that each $D_{i,j}$ corresponds to a commitment to a single bit with a value of 0 or 1, an account identifier x_i, and a nonce $r_{i,j}$ by using a standard OR-method for composing Schnorr proofs [14].

$$L_2 = PoK\left\{\begin{array}{l}(x_i, v_{i,j}, r_{i,j}) : D_{i,j} = g^{x_i} h^{v_{i,j}} f^{r_{i,j}} \\ \qquad\qquad\qquad \wedge (v_{i,j} = 1 \vee v_{i,j} = 0)\end{array}\right\}$$

1) P picks random values $b_1, b_2, z_1, z_2 \in \mathbb{Z}_q$, and c_r comprising random bits equal in number to the output of H, and then computes and sends γ_a, γ_b to V.
$$\gamma_a = \begin{cases} g^{b_1} f^{b_2} & \text{if } v_{i,j} = 0 \\ g^{z_1} f^{z_2} / D_{i,j}{}^{c_r} & \text{if } v_{i,j} = 1 \end{cases}$$
$$\gamma_b = \begin{cases} g^{z_1} h^{1+c_r} f^{z_2} / D_{i,j}{}^{c_r} & \text{if } v_{i,j} = 0 \\ g^{b_1} h f^{b_2} & \text{if } v_{i,j} = 1 \end{cases}$$
2) P generates a challenge c by using the random-oracle H and the tag 0^n.
$$c = H(g, h, f, D_{i,j}, \gamma_a, \gamma_b, 0^n)$$
3) P computes and sends $c_a, c_b, z_a, z_b, z_c, z_d$ to V.
$$c_a = c_r \oplus \overline{v_{i,j}} c$$
$$c_b = c_r \oplus v_{i,j} c$$
$$z_a = \begin{cases} b_1 + c_a x & \text{if } v_{i,j} = 0 \\ z_1 & \text{if } v_{i,j} = 1 \end{cases}$$
$$z_b = \begin{cases} b_2 + c_a r & \text{if } v_{i,j} = 0 \\ z_2 & \text{if } v_{i,j} = 1 \end{cases}$$
$$z_c = \begin{cases} z_1 & \text{if } v_{i,j} = 0 \\ b_1 + c_b x & \text{if } v_{i,j} = 1 \end{cases}$$
$$z_d = \begin{cases} z_2 & \text{if } v_{i,j} = 0 \\ b_2 + c_b r & \text{if } v_{i,j} = 1 \end{cases}$$
4) V verifies
$$g^{z_a} f^{z_b} \stackrel{?}{=} \gamma_a D_{i,j}{}^{c_a}$$
$$g^{z_c} h^{1+c_b} f^{z_d} \stackrel{?}{=} \gamma_b D_{i,j}{}^{c_b}$$
$$c_a \oplus c_b \stackrel{?}{=} c$$

When splitting challenges, it is possible to arrive at values larger than the maximum for the chosen group. Such results must be transmitted as they are, without being taken mod(q), in order for the XOR operation to remain valid. As a result, randomly generated challenges must comprise the correct number of randomly-generated bits, instead of a number randomly chosen within the group, and all challenges must be transmitted pre-modulus. Otherwise, some bits may be leaked in published proofs.

Transmission of c may be omitted, as the verifier should generate it locally. For ledger bit proofs, either c_a or c_b must be transmitted, but it is not necessary to transmit both, as one may be derived from the other and c.

D. Proof of Ledger Commitment Value Range

The verifier verifies that each C_i corresponds to a commitment to a value $0 \leq v_i < 2^\ell$ (i.e., each v_i is a small non-negative number) by testing the equality of the commitment C_i and the commitments to each of its bits, $D_{i,0} \ldots D_{i,\ell-1}$. If C_i is verifiably equivalent to the product of ℓ one-bit commitments, then the balance to which it is a commitment can comprise no

more than ℓ bits. So long as ℓ is smaller than the number of bits required to represent a negative number in group $\mathbb{G}$, then C_i cannot correspond to a negative value.

1) V computes
$$D_i = \prod_j 2^j D_{i,j}$$
and verifies
$$C_i \stackrel{?}{=} D_i$$

E. Proof of Ledger Commitment Correctness

The prover P provides the holder of each account i with an opening to the commitment to their own ledger entry. This opening comprises the account identifier x_i and balance v_i, which the account holder already knows, the index i, and the nonce r'_i. If V is the holder of account i, V may use this information to open the commitment C_i and thereby verify that v_i has been included in the sum of liabilities.

$$L_3 = PoK \left\{ \begin{array}{r} (x'_i, v_i, r'_i) : x'_i = x(2^\ell - 1) \\ \wedge v_i = v \\ \wedge r'_i = \sum_{j=0}^{\ell-1}(2^j r_{i,j}) \end{array} \right\}$$

1) If V is account holder i and expects that $x'_i = x(2^\ell - 1)$, $v_i = v$, and $r'_i = r$, then V computes
$$C'_i = g^{x(2^\ell - 1)} h^v f^r$$
and verifies
$$C_i \stackrel{?}{=} C'_i$$

Theorem 2: Under the discrete logarithm assumption and the Fiat-Shamir heuristic, the ledger commitments and ledger bit commitments are secure commitments (Definition 3), and the proof of ledger commitment value range is a non-interactive zero-knowledge proof in the random oracle model (Definition 4) for the language L.

This theorem follows immediately from Thm. 1, [16] and [14].

F. Proofs of Solvency

In this section we introduce two variants of the Proof of Solvency. Proof of Solvency by Equality enforces strict equality and requires the revelation of the total liabilities of the institution. Proof of Solvency by Inequality does not reveal the total liabilities, instead proving only that the total liabilities are less than or equal to some publicly stated value (i.e., the total assets if full reserve is required).

We intended that only one method will be employed by any real-world implementation. If Proof of Solvency by Equality is used, the difference bit commitments and proofs from the previous sections may be omitted.

Proof of Solvency by Equality. If P is willing to reveal the exact value of the total liabilities, Y, P may produce the proof that $\sum_i v_i = Y$ for some public value $Y < X$. The parties then engage in the proof system as before

$$L_{4E} = PoK \left\{ \begin{array}{r} (\sum_i x'_i, \sum_i v_i, \sum_i r'_i) : \\ C_T = g^{\sum_i x'_i} h^{\sum_i v_i} f^{\sum_i r'_i} \\ \wedge \sum_i v_i = Y \end{array} \right\}$$

1) P and V arrive at the same commitment by computing
$$C_T = \prod_i C_i$$
2) P picks a random $b_1, b_2, b_3 \in \mathbb{Z}_q$, computes γ, and sends γ to V.
$$\gamma = g^{b_1} h^{b_2} f^{b_3}$$
3) P generates a challenge c by using the random-oracle H and the tag 0^n.
$$c = H(g, h, f, C_T, \gamma, 0^n)$$
4) P computes and sends z_1, z_2, z_3 to V.
$$z_1 = b_1 + c\sum_i x'_i$$
$$z_2 = b_2 + c\sum_i v_i$$
$$z_3 = b_3 + c\sum_i r'_i$$
5) V computes
$$C'_T = C_T / h^Y$$
$$b_2 = z_2 - cY$$
$$\gamma' = \gamma / h^{b_2}$$
and then verifies
$$g^{z_1} f^{z_3} \stackrel{?}{=} {C'_T}^c \gamma'$$

Proof of Solvency by Inequality. P proves that the sum of liabilities is less than the stated assets (i.e. $\sum_i v_i < X$). This is achieved by verifying that $D_{Z,0} \ldots D_{Z,\ell-1}$ are commitments to the bits of $X - \sum_i v_i$, where X is known to V and C_T is a commitment to $\sum_i v_i$.

If C_Z is verifiably equivalent to the product of ℓ one-bit commitments, then it can comprise no more than ℓ bits. So long as ℓ is smaller than the number of bits required to represent a negative number in group $\mathbb{G}$, then the value to which C_Z is a commitment must be positive, and thus the published assets, X, must be larger than total liability $\sum_i v_i$.

$$L_{4I} = PoK \left\{ \begin{array}{r} (\sum_i x'_i, \sum_i v_i, \sum_i r'_i) : \\ C_T = g^{\sum_i x'_i} h^{\sum_i v_i} f^{\sum_i r'_i} \\ \wedge \sum_i v_i < X \end{array} \right\}$$

1) P and V arrive at the same total commitment by computing
$$C_T = \prod_i C_i$$
$$C_Z = h^X / C_T$$
2) P produces the proof that $D_{Z,j}$ corresponds to a commitment to a single bit $v_{Z,j} = 0 \vee v_{Z,j} = 1$, as in Section VII-C.
3) V computes
$$D_{Z,T} = \prod_j 2^j D_{Z,j}$$
and then verifies
$$C_Z \stackrel{?}{=} D_Z$$

VIII. Cost Analysis

Here we discuss the total computational and communication complexity for the prover and verifier. In Section IX, we describe our concrete implementation and present experimental results.

Asymptotic Analysis. Most components of the proof generation algorithm are linear in terms of ledger size or bit count. Ledger entry proofs are linear in terms of ledger size. Difference bit proofs and the final proof-of-solvency are linear in terms of bit count. Ledger Bit Proofs are linear in terms of both ledger size and bit count. Thus, the overall time complexity for proof generation and verification is $\Theta(n\ell)$, where n is the length of the ledger and ℓ is the number of bits. Space and communication complexities share this bound.

Exact Figures. We analyze the cost assuming that the caching methods discussed in Section VII-B are used, but we omit any potential advantage gained by generating proofs incrementally, as will be discussed in Section IX-B. We also assume the use of the Inequality Proof.

Table I shows the number of group exponentiations per step required to generate and verify a proof. Group exponentiations are the most expensive operations by far; other operations do not contribute significantly, and have similar complexities regardless.

Table II shows the number of elements per step publicly transmitted between the prover and the verifier (omitting the private information necessary to verify the inclusion of an individual account balance as described in Section VII-E). Our implementation uses $\ell = 24$, with each big integer requiring 256 bits and each group element requiring 258 as discussed in Section IX-A). Thus, including one-byte delimiters, each ledger entry requires 9012 bytes in a practical proof transcript. We provide an in-depth discussion of practical space and bandwidth requirements in Section IX-E.

TABLE I: Exponentiations required

Proof Component	Generate	Verify
Ledger Entry Commitment	$3n$	0
Ledger Bit Commitment	$\ell n+n$	0
Difference Bit Commitment	$\ell+1$	0
Ledger Entry Proof	$3n$	$4n$
Ledger Correctness Proof	0	3
Ledger Bit Proof	$6\ell n$	$7\ell n$
Proof-of-Solvency	6ℓ	1
Total	$7\ell n+7\ell+7n+1$	$7\ell n+4n+4$

IX. Implementation

To evaluate the concrete performance of our scheme, we built a prototype implementation of ZeroLedge, which takes the form of two programs: a proof generator, which ingests a raw ledger, performs the steps specified for the prover, and outputs a proof transcript and other relevant data, and a proof verifier, which ingests a proof transcript and optionally a commitment opener, and performs the steps specified for the verifier, indicating whether the proof is valid or not, and in what ways it may have failed.

TABLE II: Count of elements publicly transmitted

Proof Component	Group Elements	Big Integers
Ledger Entry Comm.	n	0
Ledger Bit Comm.	ℓn	0
Difference Bit Comm.	ℓ	0
Ledger Entry Proof	2	$3n$
Ledger Bit Proof	ℓn	$5\ell n$
Proof of Solvency	ℓ	5ℓ
Total	$2\ell n+2\ell+2n$	$5\ell n+5\ell+3n$

A. Prototype

We implemented our ZeroLedge prototype using C++ with the MIRACL library for big-integer math. The implementation is available under an open source license at `http://zeroledge.org`, and comprises about 2500 lines of code, excluding libraries.

We selected the Koblitz curve `secp256k1` [29] for $\mathbb{G}$, as it is widely known and supported (this is the curve used by the Bitcoin protocol for digital signatures). For random oracle H we use the MIRACL implementation of SHA-256.

Bases g, h, and f are generated from π, $\sqrt{2}$, and e. Specifically, to generate g, we interpret the first 100 decimal digits of π as an element $x_g \in \mathbb{Z}_q$, and then find the smallest $x'_g \geq x_g$ such that there exists a point $g = (x_g, y_g) \in \mathbb{G}$. Bases h and f are generated similarly from $\sqrt{2}$ and e respectively. Our hardness assumption states that it is difficult to find the discrete logarithm of h or f with respect to g and f with respect to h.

The finest subdivision of a bitcoin is the *satoshi*, equal to 10^{-8} bitcoin, and the bitcoin protocol enforces a lifetime maximum of $21 * 10^{14}$ satoshi. Thus, it requires 51 bits to represent the sum of all possible bitcoin with maximum precision. However, it is unlikely that a single entity could ever come to control this quantity. Furthermore, it is not necessary to represent the currency at its finest unit. Bitcoin transactions are limited by current policy to quantities greater than 546 satoshis [5]. Smaller transactions are typically dropped by the network. It seems unnecessary to represent any quantities smaller than the bitcoin network itself accepts for transactions, and in practice even this level of precision may be unnecessary.

Thus, we configured ZeroLedge to limit all account values to 24 bits, which is sufficient for representing account values up to 1677.7 bitcoin in units of $10,000$ satoshis, or 16777.2 bitcoin in units of $100,000$ satoshis. At the time of writing, this implies the rounding of each account to the nearest US$0.03 or $0.30, respectively.

Our implementation outputs flat-text files and uses `base64` encoding to represent big integers. Point compression and the omission of all unnecessary data and markup are used to minimize the output size.

The proof generator can be instructed to save its calculations to a separate flat-text output. This is *incremental data*, and can be used to generate subsequent incremental proofs as described in Section IX-B.

Our implementation of the proof generator saves a copy of the ledger entry data, which must be transmitted to individual

account holders in order for them to be able to verify the proof. For each account this comprises an account index, user ID, balance, and nonce.

To support parallel execution, we used a naïve threading model wherein each thread claims and processes an independent group of entries. All threads keep local products of their commitments (bitwise and overall), and when the ledger is exhausted, these products are returned to the main thread, where they are combined to form the product of commitments used in difference bit commitment generation.

B. Incremental Proofs

For any proofs subsequent to the first which share the same values of g, h, and f, it is possible to save computation time by retaining and reusing computations from the first proof. Exponentiations are particularly computationally expensive; therefore it is advantageous to replace them with other operations if possible. Here we will present a method for doing so, as it applies to a generic proof of knowledge. Our implementation includes an option to calculate subsequent proofs using this method, and we have evaluated its performance characteristics in practice, as described in Section IX-D.

We begin with a proof of knowledge

$$PoK\{(x,v,r_1) : C_1 = g^x h^v f^{r_1}\}$$

and show how to transform it into a new proof of knowledge with a new nonce r_2 (the value, v, may be changed in a similar way):

$$PoK\{(x,v,r_2) : C_2 = g^x h^v f^{r_2}\}$$

1) P recalls the original proof, specifically
   a) the public commitment C_1
   b) an account identifier, account value, and nonce, x, v, r_1 respectively
   c) random values b_1, b_2, b_3
   d) one-time commitment γ_1
2) P picks a random $r_2 \in \mathbb{Z}_q$, and calculates nonce r_3.
$$r_3 = r_2 - r_1$$
3) P calculates and publishes C_2 as its new public commitment.
$$C_2 = C_1 f^{r_3}$$
4) P picks a random $b_4 \in \mathbb{Z}_q$, computes γ_2, and sends γ_2 to V.
$$\gamma_2 = \gamma_1^{b_4}$$
5) P generates a challenge c_2 by using the random-oracle H and the tag 0^n.
$$c_2 = H(g,h,f,C_2,\gamma_2,0^n)$$
6) P computes and sends (z_4, z_5, z_6) to V.
$$z_4 = b_1 b_4 + c_2 x$$
$$z_5 = b_2 b_4 + c_2 v$$
$$z_6 = b_3 b_4 + c_2 r_2$$
7) V verifies
$$g^{z_4} h^{z_5} f^{z_6} \stackrel{?}{=} {C_2}^{c_2} \gamma_2$$

The application of this method to ledger entry commitments as outlined in Section VII-C is straightforward. For ledger bit proofs, the one-time commitment for the incorrect value must be recomputed completely; meanwhile, if the correct bit value is 1, its commitment must be divided by h before being taken to the power of b_4 in step 4, then multiplied by h again afterward.

Cost. In Step 3 we save two exponentiations, in comparison to generating a new commitment from scratch. In Step 4, two additional exponentiations are saved. However, this method requires the storage of relevant data for reuse.

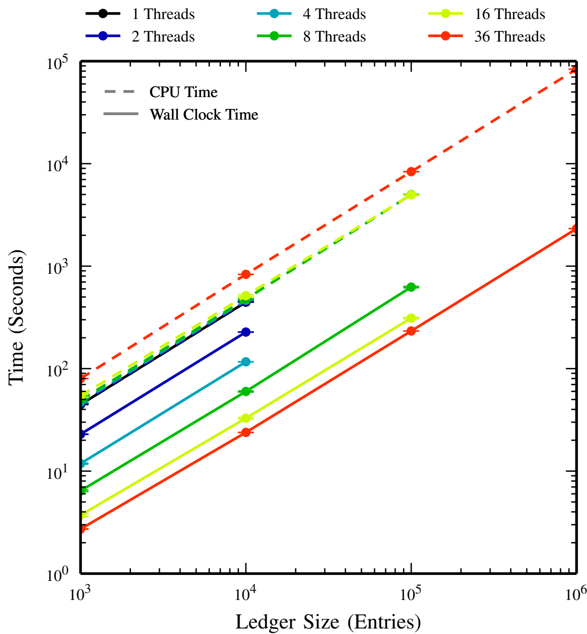
C. Experimental Setup

All tests were run on an Amazon EC2 `c4.8xlarge` node, containing two Intel Xeon E5-2666v3 processors, clocked at 2.9 GHz, and 60 GB of ECC DDR4 memory. Each E5-2666v3 has ten cores which share 25MB of L2 cache; each core is capable of executing two threads. Because, only 18 physical cores are exposed to the user by Amazon, the maximum number of simultaneous threads is 36. Tests were run under Amazon Linux (kernel 3.14.44, 64 bit), and the code was compiled using GCC 4.8.2 with the `-O2` flag. MIRACL was configured to use Comba multiplication and inline assembly.

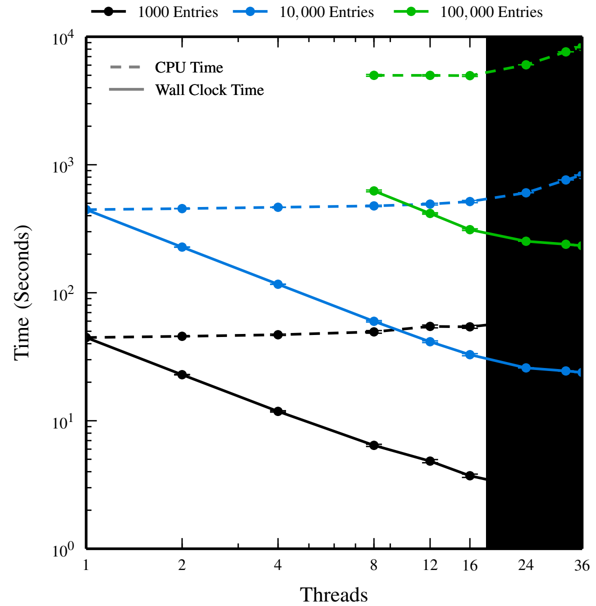
Benchmarks were performed using a shell script which repeatedly executed the proof generator, creating a new input ledger on each iteration. Entries in this ledger each comprise a random, eight character identifier and a balance between 0 and 1000, inclusive. The unix `time` program was used to measure the wall clock time and total CPU time elapsed during the generation of each proof.

On each iteration of the benchmark script, five different measurements were taken. First, a proof was generated from scratch, exporting only the proof and ledger data. Second, an identical proof was generated, exporting incremental data in addition. Third, a modified proof generator ingested the incremental data and then terminated; the time spent by this program is representative of the ingest time for incremental data in the subsequent two steps. Fourth, a proof was generated incrementally, using the original ledger and the incremental data exported by the proof generator in step two (and exporting its own incremental data). This represents the best case incremental proof, in which no account balances have changed and the maximum amount of work can be reused. Finally, another proof was generated incrementally, this time using a ledger with *new* randomly generated balances and the incremental data exported by the proof generator in step two (also exporting its own incremental data). This represents the worst case incremental proof, in which all account balances have changed and the minimum amount of work can be reused.

We benchmarked proof verification time using a similar method. For each iteration, our benchmark script generated a new random ledger and a new proof, and then measured two different verification processes. First, the proof was verified fully. Second, one of the ledger entries was selected at random, and only the inclusion of this particular entry was verified. This second process represents the minimum effort each account holder must make in order to ensure that absolutely no account can be fraudulently reduced or omitted.



Values are averages of 30 samples, stated with a 95% confidence interval for all ledger sizes except 1,000,000. Values for ledgers of size 1,000,000 are averages of 3 samples, stated with a 95% confidence interval.



Values are averages of 30 samples, stated with a 95% confidence interval. Shaded region represents system load of more than one thread per physical core.

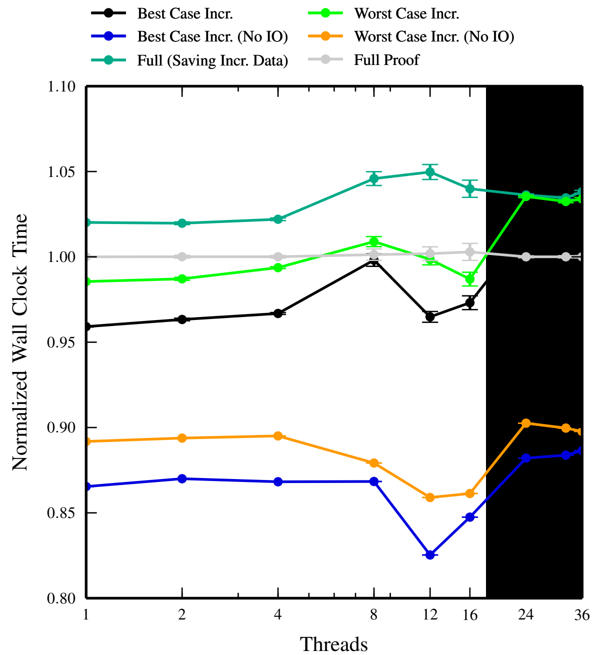
D. Experimental Results: Time

For proof generation, we benchmarked four different ledger sizes and several levels of parallelism. Runtimes are reported in Figure 4 against ledger size. Our data reveals that ledger size has a linear relation to both CPU and wall clock time, as expected.

Figure 5 shows runtimes against thread count. With a loading of less than one thread per core, CPU time increases slightly with thread count, due to the overhead required to coordinate a greater number of threads, and wall clock time decreases linearly with thread count. With a loading of greater than one thread per core, CPU time increases significantly. Wall clock time continues to improve as the thread count increases beyond the 18 available cores, but at a lesser rate.

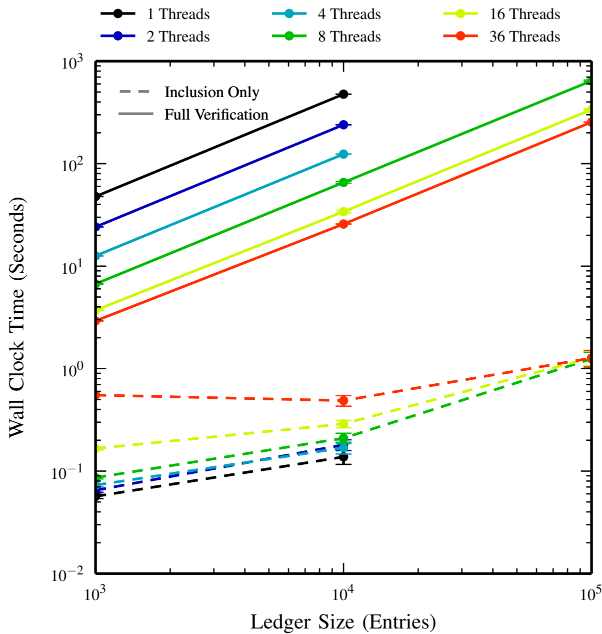
In theory, it is possible for a proof generator using the same ledger-splitting parallelism as our own to operate with no inter-thread communication at all, beyond the initial splitting of the ledger and the pooling of results. Such an implementation could be scaled to an arbitrary number of physical cores or machines.

The relative computation times for incremental proofs reported in Figure 6 are less straightforward. Times marked with *No IO* are calculated to simulate the generation time of an incremental proof without penalties due to ingest or export of incremental data. This is indicative of the performance of a proof generator which holds its data in memory between proofs, although our implementation does not actually work in this way. Incremental data export times are calculated by subtracting the standard full proof time from the full proof time with incremental data export. The *No IO* incremental proof



Values are averages of 30 samples, stated with a 95% confidence interval. Shaded region represents system load of more than one thread per physical core. All data is taken from samples with ledger size 10,000. Wall clock times are normalized by Full Proof Wall Clock Time.

Fig. 7: Ledger Size vs Verification Time



Values are averages of 30 samples, stated with a 95% confidence interval.

times are then calculated by subtracting the incremental data export and incremental data ingest times from the incremental proof times.

For any thread count, the computation time of a best case incremental proof is always better than that of a worst case incremental proof, but in the worst case incremental proofs do not significantly improve upon full proofs generated from scratch. In no case do the incremental proofs improve upon full proof generation by more than 5% when including IO penalties. Omitting the IO penalties increases the performance of incremental proofs significantly; even in the worst case, they save 10% of the wall clock time, relative to the full proof.

The computation time for a single ledger entry on a single core is 49.9ms on our machine for a full proof, or 42.1ms for best-case incremental with no IO (based upon samples for 8 threads and 100,000 entries). This works out to a computation rate of about 72140 and 85510 ledger entries per hour, per thread, respectively, assuming one thread per core.

The Amazon EC2 `c4.8xlarge` node on which we conducted our benchmarks currently costs US$1.763 per hour. With 36 threads active, it requires 38.7 minutes to process one million entries. Thus, the cost of generating a full proof for a ledger with one million entries is US$1.09.

For proof verification, we benchmarked three different ledger sizes and several levels of parallelism. Runtimes are reported in Figure 7 against ledger size. Our data reveals that ledger size has a linear relation to full verification wall clock time, as expected.

Although we verify the inclusion of only one entry regardless of ledger size in our inclusion-only benchmark, verification time does increase with ledger size due to the fact that longer proofs take more time to scan. A more advanced verifier than ours might predict the location of the one relevant entry and seek there directly, yielding constant verification-of-inclusion time regardless of ledger size.

E. Experimental Results: Bandwidth

As the output length is consistent and directly dependent on the input ledger length, sizes are exactly consistent over all trials. Reported in Table III are the character (i.e. byte) counts for each individual component of the output. Using this data, the proof size for any number of bits and ledger entries can be predicted.

Table IV contains actual proof and data sizes for randomly generated ledgers of various lengths. As before, each ledger entry is configured to include 24 bit proofs. Ledger data represents the size of the input into the proof generator: an index, balance, identifier, and nonce for each account. Incremental data must be stored (either on disk or in memory) only if subsequent incremental proofs are to be generated. All outputs are gzipped to improve efficiency. The relationship between ledger entries and output size is almost exactly linear, regardless of output type.

TABLE III: Component-wise size of a proof, in bytes

Component	Size
Header	102
Bases	141
Ledger Entry Proof	229
Ledger Bit Proof	366
Difference Bit Proof	366
Footer/Misc	61

Values are stated without compression. Our implementation uses 24 bits, and so each ledger entry requires a total of 9012 bytes, including delimiters.

TABLE IV: Size of proofs and data, in megabytes

Ledger Size	Proof	Ledger Data	Incr. Data
1000	6.4	0.05	4.1
10,000	64.41	0.48	41.31
100,000	644.06	4.70	413.12

X. Conclusion

ZeroLedge provides a practical solution to private proof-of-liability, which overcomes the drawbacks of the Maxwell Protocol. For account holders, it increases privacy by providing anonymous verification and eliminating leakage of account details. It also reduces the avenues for cheating available to a fraudulent institution, by preventing the prover from tracking the verifiers, reducing the degrees of freedom in proof generation, and permitting all verifiers to individually check proof integrity. For financial institutions, it reduces the information leakage inherent in the existing system and removes the burden of interaction with verifiers.

While we focus on cryptocurrency in general, and bitcoin in specific, ZeroLedge has applications beyond cryptocurrency

exchanges as well. It is not tied to cryptocurrency in any way, and should work just as effectively for institutions which hold assets on behalf of clients in any other form, provided that there is a way to guarantee the quantity of assets actually held. While it is not a replacement for audits or traditional mechanisms of accountability, ZeroLedge might serve as a supplementary assurance for customers of traditional financial institutions. We think this idea may also be amenable to the institutions themselves, as it implies the release of much less data than they are already required to divulge by law.

Fractional reserve institutions may find ZeroLedge additionally valuable as a means to conceal the exact fraction currently held. Because it proves only that the the total liability is less than a particular published number X, a fractional reserve may set X to any value which is both larger than the sum of their liabilities and smaller than the quotient of their published reserved assets and reserve fraction. X may, for instance, be pegged by policy to the largest valid value. Used in this way, ZeroLedge reveals only that the institution holds at least the minimum allowable fraction; it does not reveal the true fraction held, nor does it require the publication of the actual total assets of the institution.

REFERENCES

[1] Securities exchange act of 1934, pub. l. no. 73-291, §13, 1934.

[2] Federal deposit insurance act, 12 u.s.c. §1817(a)(1), 1950.

[3] Bitcoin transaction 29a3efd3ef04f9153d47a990bd7b048a 4b2d213daaa5fb8ed670fb85f13bdbcf, 2011.

[4] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *USENIX Security 2014*, 2014.

[5] Bitcoin Foundation. Non-standard transactions. https://bitcoin.org/en/developer-guide#non-standard-transactions, 2015.

[6] Bitfoo, Ltd. Proof of reserves. https://github.com/bifubao/proof_of_reserves, March 2014.

[7] Board of Governors of the Federal Reserve System. Reserve requirements. http://www.federalreserve.gov/monetarypolicy/reservereq.htm, 2015.

[8] Jonas Borchgrevink. Bifubao unveils first production-scale proof of reserves. https://www.cryptocoinsnews.com/bifubao-unveils-first-production-scale-proof-reserves/, March 2014.

[9] Fabrice Boudot. Efficient proofs that a committed number lies in an interval. In *Advances in Cryptology — EUROCRYPT '00*, 2000.

[10] Ken Brown and Ianthe Jeanne Dugan. Arthur andersen's fall from grace is a sad tale of greed and miscues. Wall Street Journal, June 2012.

[11] Jan Camenisch, Rafik Chaabouni, and Abhi Shelat. Efficient protocols for set membership and range proofs. In *Advances in Cryptology — ASIACRYPT '08*, 2008.

[12] Rafik Chaabouni, Helgar Lipmaa, and Abhi Shelat. Additive combinatorics and discrete logarithm based range protocols. In *15th Australasian Conference on Information Security and Privacy (ACISP)*, 2010.

[13] Caleb Chen. Bitfinex passes stefan thomas's proof of solvency audit. https://www.cryptocoinsnews.com/bitfinex-passes-stefan-thomass-proof-solvency-audit/, July 2014.

[14] Ronald Cramer, Ivan Damgard, and Berry Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In *Advances in Cryptology — CRYPTO '94*, 1994.

[15] Christian Decker, James Guthrie, Jochen Seidel, and Roger Wattenhofer. Making bitcoin exchanges transparent. In *20th European Symposium on Research in Computer Security (ESORICS)*, 2015.

[16] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Advances in Cryptology — CRYPTO '86*, 1987.

[17] Eiichiro Fujisaki and Tatsuaki Okamoto. Statistical zero knowledge protocols to prove modular polynomial relations. In *Advances in Cryptology — CRYPTO '97*, 1997.

[18] Jens Groth. Non-interactive zero-knowledge arguments for voting. In *Applied Cryptography and Network Security (ACNS)*, 2005.

[19] Jens Groth and Amit Sahai. Efficient non-interactive proof systems for bilinear groups. In *Advances in Cryptology — EUROCRYPT '08*, 2008.

[20] Olivier Lalonde. Proof of liabilities. https://github.com/olalonde/proof-of-liabilities, March 2014.

[21] Helger Lipmaa. On diophantine complexity and statistical zero-knowledge arguments. In *Advances in Cryptology — ASIACRYPT '03*, 2003.

[22] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology — CRYPTO '87*, 1988.

[23] Brian Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, 2013.

[24] Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Advances in Cryptology — CRYPTO '91*, 1992.

[25] Jesse Powell. Jespow comments on please ask your favorite exchange to prove that they are not running a fractional bitcoin reserve. http://www.reddit.com/r/Bitcoin/comments/1yk4nv/please_ask_your_favorite_exchange_to_prove_that/cflqtn0, February 2014.

[26] C. P. Schnorr. Efficient identification and signatures for smart cards. In *Advances in Cryptology — EUROCRYPT '89*, 1990.

[27] Berry Schoenmakers. Some efficient zeroknowledge proof techniques. Slides presented at the *International Workshop on Cryptographic Protocols*, March 2001. Monte Verita, Switzerland.

[28] Berry Schoenmakers. Interval proofs revisited. Slides presented at the *International Workshop on Frontiers in Electronic Elections*, September 2005. Milan, Italy.

[29] Standards for Efficient Cryptography Group. Sec 2: Recommended elliptic curve domain parameters. http://www.secg.org, January 2010.

[30] Stefan Thomas. Easy audit. https://github.com/justmoon/easy-audit, August 2014.

[31] Zak Wilcox. Proving your bitcoin reserves. https://iwilcox.me.uk/2014/proving-bitcoin-reserves, February 2014.