MiniLedger: Compact-Sized Anonymous and Auditable Distributed Payments

Panagiotis Chatzigiannis(✉) and Foteini Baldimtsi

George Mason University, Fairfax, USA
{pchatzig,foteini}@gmu.edu

Abstract. In this work we present MiniLedger, a distributed payment system which not only guarantees the privacy of transactions, but also offers built-in functionalities for various types of audits by any external authority. MiniLedger is the *first* private and auditable payment system with storage costs independent of the number of transactions. To achieve such a storage improvement, we introduce pruning functionalities for the transaction history while maintaining integrity and auditing. We provide formal security definitions and a number of extensions for various auditing levels. Our evaluation results show that MiniLedger is practical in terms of storage requiring as low as 70 KB per participant for 128 bits of security, and depending on the implementation choices, can prune 1 million transactions in less than a second.

1 Introduction

One of the main issues with distributed ledger-based (or else blockchain) payment schemes (e.g. Bitcoin) is the lack of privacy. All transaction information - including transacting parties' public keys and associated values - are permanently recorded on the public blockchain/ledger, and using side-channel information these keys can be clustered and eventually linked to real identities [5,30]. A number of solutions have been suggested to solve the privacy issues of distributed ledgers by hiding both the transaction graph and its associated assets and amounts [7,14,19,39], however, while privacy is a fundamental right, the need for *auditing mechanisms* is required to ensure compliance with laws and regulation [1,24] as done in traditional payment systems via auditing companies (i.e. Deloitte, KPMG). Constructing payment schemes that satisfy both privacy and auditability *at the same time*, is a rather challenging problem since these properties are often conflicting. The challenge becomes even harder when one takes *efficiency and scalability* into account. In particular, one of the most common approaches to solve the scalability issue, that of pruning old/unneeded transactions from the ledger, directly hurts auditability, as an auditor cannot possibly audit data that no longer exists in the ledger.

F. Baldimtsi—The authors have been supported by the National Science Foundation (NSF) under Grant 1717067, the National Security Agency (NSA) under Grant 204761, an IBM Faculty Award and Facebook Research Award.

E. Bertino et al. (Eds.): ESORICS 2021, LNCS 12972, pp. 407–429, 2021.
https://doi.org/10.1007/978-3-030-88418-5_20

While a number of solutions for accountable and private distributed payments have been proposed in the literature, they either rely on the existence of trusted authorities or do not scale for large number of participants and transactions.

Our Contributions. We present MiniLedger: the *first space efficient*, distributed private payment system that allows an authorized set of participants to transact with each other, while also allowing for a wide set of auditing by consent operations by *any* third party auditor. We provide formal, game based definitions (in full version [15]) and a construction that relies upon a number of cryptographic primitives: a consensus protocol, semi-homomorphic encryption, compact set representation techniques (cryptographic accumulators) and non-interactive zero-knowledge proofs (NIZKs).

At a high level, MiniLedger consists of n Banks transacting with each other through a common transaction history, or else a ledger L which is maintained by a consensus mechanism (orthogonal to our work). The ledger is modeled as a two-dimensional table with n columns, one for each participating Bank, and rows representing transactions. Whenever Bank B_j wishes to send funds of value v to another B_k, it creates a n-sized vector containing (semi)homomorphic encryptions and NIZK proofs which is appended in L. B_j encrypts the value that is sent to each participating Bank in the system using each receiving Bank's public key, i.e. the encrypted values would be v for B_k, $-v$ for B_j and 0 for any other Bank. These encryptions provide privacy in MiniLedger since they hide values as well as the sender and recipient of each transaction, while still allowing all participating Banks to decrypt the value that corresponds to them and to compute their total assets at any point. This overcomes the need for any out-of-band communication between Banks which created security issues in previous works (ref. Sect. 4.2). Finally, the included ZK proofs guarantee that transactions are valid without revealing any information.

MiniLedger provides auditability *by consent*. Any third party auditor with access to L can ask audit queries to a Bank and verify the responses based on the public information on L. The simplest audit is to learn the value of a cell in L, i.e. the exact amount of funds a Bank received/sent at any point. This basic audit can be used to derive more complex audit types as we discuss in Appendix A, such as transaction history, account balance, spending limit etc., without disclosing more information to the auditor than needed.

Space Efficiency. The main innovation of MiniLedger lies in the maintenance and storage of L. In previous auditable schemes (such as zkLedger [34]) the *full* L needs to be stored at *any time* and by *all* participants. The challenge in MiniLedger design was compacting the ledger while maintaining security and a wide set of auditing functionalities. MiniLedger employs a smart type of transaction pruning: participating Banks can prune their own transaction history by computing a provable, *compact representation* of their previously posted history and broadcast the resulting digest to the consensus layer. Once consensus participants agree to a pruning operation (i.e. verify the digest as a valid representation of the Bank's history), that history can be erased from L and thus by all system participants (except the pruning Bank itself which always need to store its own transaction history locally). Auditing is still possible since a compact digest of transaction information is always stored in L and the Bank under audit can prove that the revealed values correctly correspond to the digest. As a result, the size of L in MiniLedger can be nearly constant (i.e. independent of the number of transactions that ever happened).

Our compact transaction history representation can lead to multiple additional benefits (besides obviously reduced storage requirements). First, a compact L makes addition of new system participants (i.e. Banks) much more efficient (typically, new parties need to download the whole L requiring large bandwidth and waiting time). Then, although the structure of L does not allow for a very large number of participating Banks n (as the computation cost of a single transaction is linear in n), the compactness of L allows augmenting MiniLedger with more fine-grained types of auditing and enabling audits in a client level (instead of a Bank level). We present MiniLedger+, an extension that serves a much larger user base in Appendix A.

Finally, we implement a prototype of the transaction layer of MiniLedger and evaluate its performance in terms of transaction costs, pruning costs and size of L which we estimate to be as low as 70KB of storage for each Bank. We show that transaction computation cost, for a system with 100 Banks, takes about 4 sec, while transaction auditing is less than 5 ms, independent of the number of Banks. Transaction computation costs increase linearly to the number of Banks (as in zkLedger) but by optimizing the underlying ZK proofs we achieve some small constant improvement. Although the linear transaction computation cost might still seem high, we note that using our MiniLedger+ extension, a small number of Banks can serve a very large user base. We perform experiments on two different types of pruning instantiations, one using Merkle trees [31] and one using batch RSA accumulators [8]. Both result in pruning measurements that are independent of the number of participating Banks. Our experiments show that we can prune 1 million transactions in less than a second using Merkle trees and in about 2 h using the RSA accumulator, and can perform audits in milliseconds in the same transaction set.

Related Work. We present an non-exhaustive overview of related works.

Anonymous Distributed Payment Systems. Zcash [7], is a permissionless protocol hiding both transacting parties and transaction amounts using zero knowledge proofs. Other notable systems are CryptoNote and the Monero cryptocurrency [39], based on decoy transaction inputs and ring signatures to provide privacy of transactions within small anonymity sets, and Quisquis [19] which provides similar anonymity level to Monero but allows for a more compact sized ledger. Zether [10] is a smart contract based payment system which only hides transaction amounts. Mimblewimble [20] uses Confidential Transactions [28] to hide transaction values in homomorphic commitments, and prunes intermediate values from the blockchain after being spent (which might be insecure in

Table 1. Confidential payment schemes comparison. By ✔^{S} we denote set anonymity, ✔^{T} auditing through a TP and ✔^{K} through "view keys" (which reveal all private information of an account). By O: permissionless and C: permissioned we refer to the set of parties that participate in the payment scheme and not the underlying consensus.

	Record	Anon	Audit	Perm	Prune
Zcash [7,22]	UTXO	✔	✔^{T}	O	×
Monero [26,39]	TXO	✔^{S}	✔^{K}	O	×
Quisquis [19]	Hybrid	✔^{S}	×	O	✔
MW [20]	UTXO	×	×	O	✔
Solidus [14]	Accnt	✔^{S}	×	C	×
zkLedger [34]	Accnt	✔	✔	C	×
PGC [17]	Accnt	×	✔	O	×
Zether [10]	Accnt	Option	×	O	×
MiniLedger	Accnt	✔	✔	C	✔

other UTXO systems such as Bitcoin), improving its scalability. In the permissioned setting, Solidus [14] allows for confidential transactions in public ledgers, employing Oblivious RAM techniques to hide access patterns in publicly verifiable encrypted Bank memory blocks. This approach enables users to transact in the system anonymously using Banks as intermediaries.

Adding Auditability/Accountability. A number of Zcash extensions [22,25,32] proposed the addition of auxiliary information to coins to be used exclusively by a designated, trusted authority for accountability purposes. While this allows for a number of accountability functionalities, it suffers from centralization of auditing power. Additionally, all such works inherit the underlying limitations of Zcash such as the need for trusted setup and strong computational assumptions. Traceable Monero [26] attempts to add accountability features on top of Monero. In a similar idea to Zerocash, a designated "tracing" authority can link anonymous transactions with the same spending party and learn the origin or destination of a transaction. The tracing authority's role can again be distributed among several authorities to prevent single point of failure and distribute trust. PRCash [41] aims to achieve accountability for transaction volume over time. A regulation authority (can be distributed using threshold encryption) issues anonymous credentials to the system's transacting users. If transaction volume in a period exceeds a spending limit, the user can voluntarily deanonymize himself to the authority to continue transacting. PRCash however only focuses on this specific audit type. zkLedger presented a unique architecture for implementing various interactive audit types in a permissioned setting, but its linear-growing storage requirements in terms of transactions make it unpractical for real deployment. Additionally, it assumes transaction values are communicated out-of-band, creating an attack vector that could prevent participants from answering audits(further discussed in our full version [15]).

In Table 1 we summarize properties of private payment schemes and refer the reader to [16] for a systematization of knowledge on auditable and accountable distributed payment systems.

Prunable and Stateless Blockchains. Given the append-only immutability property for most ledgers, the concern for ever-growing storage requirements in blockchains was stated even in the original Bitcoin whitepaper [33], which considered *pruning* old transaction information without affecting the core system's properties. Ethereum [40], being an account-based system, supports explicit support of "old state" pruning as a default option, and defers to "archival" nodes for any history queries. Coda (Mina) [9] is a prominent example of a stateless (succinct) blockchain, which only needs to store the most recent state with recursive verifiability using SNARKs. Accumulators and vector commitments have also been proposed to maintain a stateless blockchain [8,18]. All such approaches however might negatively impact auditability and are therefore not directly applicable in our setting.

2 Preliminaries

By λ we denote the security parameter and by $z \leftarrow \mathcal{Z}$ the uniformly at random selection of an element z from space $\mathcal{Z}$. By $(\mathsf{pk}, \mathsf{sk})$ we denote a public-private key pair and by $[x_i]_{i=1}^{y}$ a list of elements $(x_1, x_2, ..., x_y)$. By $x \parallel y$ we denote concatenation of bit strings x and y. We denote a matrix M with m rows and n columns as M_{mn} and a $i-$th row and j-th column cell in the matrix as (i, j).

ElGamal Encryption Variant. MiniLedger uses a variant of ElGamal encryption (called "twisted ElGamal" (TEG) [17]). Compared to standard ElGamal, it requires an additional group generator (denoted by h below) in the public parameters pp, which makes it possible to homomorphically add ciphertexts c_2 and c_2' generated for *different* public keys pk and pk' and intentionally leak information on the relation of encrypted messages m and m' as we discuss below.

TEG is secure against chosen plaintext attacks and works as follows:

- $\mathsf{pp} \leftarrow \mathsf{SetupTEG}(1^\lambda)$: Outputs $\mathsf{pp} = (\mathbb{G}, g, h, p)$ where g, h are generators of cyclic group $\mathbb{G}$ of prime order p.
- $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{GenTEG}(\mathsf{pp})$: Outputs $\mathsf{sk} \leftarrow \mathbb{Z}_p$, $\mathsf{pk} = h^{\mathsf{sk}}$.
- $(c_1, c_2) \leftarrow \mathsf{EncTEG}(\mathsf{pk}, m)$: Sample $r \leftarrow \mathbb{Z}_p$, compute $c_1 = \mathsf{pk}^r, c_2 = g^m h^r$ and output $C = (c_1, c_2)$
- $m \leftarrow \mathsf{DecTEG}(\mathsf{sk}, (c_1, c_2)$: Compute $g^m = c_2 / c_1^{(1/\mathsf{sk})}$ and recover m from a look-up table (assuming that the message space is relatively small).

TEG encryption is additively homomorphic: $\mathsf{EncTEG}(\mathsf{pk}, m_1)\mathsf{EncTEG}(\mathsf{pk}, m_2) = \mathsf{EncTEG}(\mathsf{pk}, m_1 + m_2)$. Also if $(c_1, c_2) \leftarrow \mathsf{EncTEG}(\mathsf{pk}, m)$ and $(c_1', c_2') \leftarrow \mathsf{EncTEG}(\mathsf{pk}', m')$, then $c_2 c_2'$ contains an encryption of $m + m'$. This implies if $c_2 c_2' = 1$, then any external observer can deduce that $m = -m'$ (for properly chosen r, r').

Commitment Schemes. We use Pedersen commitments [36] which are additively homomorphic and allow efficient zero-knowledge proofs, are perfectly hiding and computationally binding and consist of the following algorithms:
(a) $\mathsf{ComGen}(1^\lambda)$ outputs $\mathsf{pp} = (\mathbb{G}, g, h, p)$ where g, h are generators of cyclic group $\mathbb{G}$ of prime order p, (b) $\mathsf{Com}(\mathsf{pp}, m, r)$ for a message $m \in [1...p]$ and randomness $r \in [1...p]$, outputs a commitment $\mathsf{cm} = g^m h^r$, (c) $\mathsf{Open}(\mathsf{pp}, \mathsf{cm}, m, r)$ with verifier given commitment cm and opening (m, r) returns verification bit b.

Zero-Knowledge Proofs. A zero-knowledge (ZK) proof is a two-party protocol between a prover P, holding some private data (or else *witness*) w for a public instance x, and a verifier V. The goal of P is to convince V that some property of w is true i.e. $R(x, w) = 1$, for an NP-relation R, without V learning anything more. In MiniLedger we use non-interactive ZK proofs (NIZKs) and OR and AND compositions of them The types of ZK proofs used in MiniLedger are:

1. ZK proof on the opening of a commitment: Using Camenisch-Stadler notation [12] (used throughout the paper): $ZKP : \{(w, r) : X = g^w h^r \bmod n\}(X, g, h, n)$ where (X, g, h, n) are the public statements given as common input to both parties, and (w, r) is the secret witness.
2. ZK proof of knowledge of discrete log: $ZKP : \{(x) : X = g^x \bmod n\}(X, g, n)$.
3. ZK proof of equality of discrete logs: $ZKP : \{(x, r, r') : X = g^x h^r \bmod n, Y = g^x h^{r'} \bmod n\}(X, Y, g, h, n)$.
4. ZK range proof that a committed value v lies within a specific interval (a, b): $ZKP : \{(v, r) : X = g^v h^r \bmod n \wedge v \in (a, b)\}(X, g, h, n)$. Such proofs can also be used to show that the value v is positive or does not overflow some modulo operation. Known range proof constructions include [11,29,37].

Cryptographic Accumulators. Accumulators allow the succinct and binding representation of a set of elements S and support constant-size proofs of (non) membership on S. We focus on *additive* accumulators where elements can be added over time to S and to *positive* accumulators which allow for efficient proofs of membership. We consider *trapdoorless* accumulators in order to prevent the need for a trusted party that holds a trapdoor and could potentially create fake (non)membership proofs. Finally we require the accumulator to be *deterministic*, i.e. always produce the same representation given a specific set. An accumulator typically consists of the following algorithms [6]:

- $(\mathsf{pp}, D_0) \leftarrow \mathsf{AccSetup}(\lambda_{acc})$ generates the public parameters and instantiates the accumulator initial state D_0.
- $\mathsf{Add}(D_t, x) := (D_{t+1}, \mathsf{upmsg})$ adds x to accumulator D_t, which outputs D_{t+1} and upmsg which enables witness holders to update their witnesses.
- $\mathsf{MemWitCreate}(D_t, x, S_t) := w_x^t$ Creates a membership proof w_x^t for x where S_t is the set of elements accumulated in D_t. $\mathsf{NonMemWitCreate}$ creates the equivalent non-membership proof u_x^t.
- $\mathsf{MemWitUp}(D_t, w_x^t, x, \mathsf{upmsg}) := w_x^{t+1}$ Updates membership proof w_x^t for x after an element is added to the accumulator. $\mathsf{NonMemWitUp}$ is the equivalent algorithm for non-membership.

- $\mathsf{VerMem}(D_t, x, w_x^t) := \{0,1\}$ Verifies membership proof w_x^t of x in D_t.

The basic security property of accumulators is *soundness* which states that for every element *not* in the accumulator it is infeasible to prove membership.

We utilize two types of accumulators: (a) the additive, universal RSA accumulator [8] and (b) additive, positive Merkle Trees [31]. We note that RSA accumulator can become trapdoorless if a trusted party (or an MPC protocol) is used to compute the primes for the modulo n, or a public RSA challenge number (i.e. from RSA Labs) is adopted. We also note that we will apply batching techniques in element additions and membership proofs [8]. In Sect. 5 we discuss the trade-offs between the two options for different implementation scenarios.

Consensus. A consensus protocol (denoted by CN) allows a set, S_{CN}, of distributed parties to reach agreement in the presence of faults. For MiniLedger we assume that the agreement is in regards to data posted on a ledger L and participation in the consensus protocol can be either *permissioned* (i.e. only authenticated parties have write access in the ledger) or *permissionless*. Consensus protocols that maintain such a fault-tolerant ledger and their details (e.g. participation credentials, incentives, sybil attack prevention etc.) are out of the scope of this paper and can be done using standard techniques [4,13]. For our construction, we assume a consensus protocol: $\mathsf{Conscus}(x, L) := L'$ which allows all system participants given some input value x and ledger state L, to agree on a new ledger L'. We assume that consistency and liveness [21] are satisfied.

3 MiniLedger Model

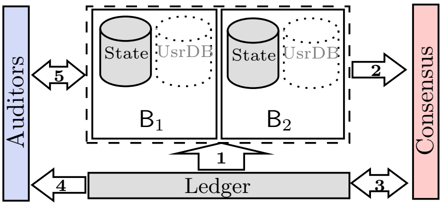


Fig. 1. MiniLedger overview. State: Bank's private database. Usr DB: optional private database for Bank's clients. Banks read Ledger to create/prune transactions (1), forward output to Consensus (2), which verifies/updates the Ledger (3). Auditor reads Ledger (4) and interacts with Banks to audit transactions (5).

We consider the following system participants: a Trusted Party TP, a set of consensus participants S_{CN}, a static set of n Banks with IDs defined by $[\mathsf{B}_j]_{j=1}^n$ (known to everyone) and an arbitrary number of Auditors A. Each Bank has a key pair $[(\mathsf{pk_j}, \mathsf{sk_j})]_{j=1}^n$ and an initial asset value $[v_j]_{j=1}^n$. Banks maintain an internal state $[st_j]_{j=1}^n$. We denote transactions by tx_i where i represents the transaction's index. We store transactions in a public ledger L maintained by a consensus layer CN and stored by all banks. We summarize the role of each participant in MiniLedger and provide the architecture overview in Fig. 1:

- TP is a trusted entity which runs an one-time setup to instantiate the system public parameters and verifies the initial assets of each Bank. TP could be replaced by an MPC protocol (i.e. [23]) executed by the Banks.
- Banks generate transactions tx that transfer some of their assets to one or more other Banks, while hiding the value and the transacting parties. Transactions are sent to the consensus layer CN (via an anonymous communications protocol, i.e. Tor) and if valid are appended on L. Banks can prune their transaction history on L and "replace" it by a digest D. The pruning Bank needs to send D to CN (incentives for the Bank to perform the pruning operation are orthogonal to our construction) and is responsible to keep a copy of the pruned transactions in its private database "State" (failing to do so can lead to audit failures). If D is valid, CN participants update L by deleting the pruned transactions and replacing them by D.
- Auditors by observing the ledger, can audit the Banks at any point for any set of transactions through interactive protocols. Auditors should be able to audit the value of a single transaction or a subset of transactions, whether these transactions are still in L or have been pruned.

Assumptions. We focus on the transaction layer and consider issues with underlying consensus and network layers and their mitigations orthogonal to this work. Specifically, we assume the fundamental consensus properties, as defined in Sect. 2, always hold. On network level, we assume a malicious Bank cannot block another Bank's view (Eclipse attacks) and transactions are sent to *all* Banks using anonymous communication channels to preserve anonymity of the sending and the receiving Bank(s). We *do not* require out-of-band communication between Banks. For simplicity, we assume the set of participating Banks is static but is easy to extend our system to dynamically add/remove Banks. We also assume the Random Oracle model to convert our ZK proofs to non-interactive.

Security Goals. MiniLedger should satisfy the following properties (formally defined in a game-based fashion in the full version [15]). *Theft prevention and balance:* When a Bank spends, we require that a) it authorized the transaction, b) its balance decreases by the appropriate amount and c) it cannot spend more than its total assets. *Secure pruning:* Ledger pruning outputs a digest that a) is only created by the respective Bank, b) contains the correct transactions in the correct order, and c) does not contain bogus transactions. *Ledger correctness:* The ledger only accepts valid transactions and pruning operations. *Correct and Sound Auditability:* An honest Bank following the protocol can always answer audits correctly and convince an Auditor, while an Auditor always rejects false claims from a malicious Bank. *Privacy:* The ledger hides both the identities of transacting parties and values of transactions from any external observer (except auditors who learn specific information during the auditing protocol).

4 MiniLedger Construction

Overview. We consider n Banks that transact with each other in an anonymous and auditable way by posting data in a common ledger L (a two-dimensional table with n columns, one for each participating Bank, and a number of rows which represent transactions). The ledger is maintained by consensus participants, who verify every submitted transaction, and is stored by all Banks. The Banks could be running consensus themselves, or outsource this operation to any external set of consensus parties.

For each tx_i, the sending Bank (i.e. the transaction creator) creates a whole row in L which includes twisted ElGamal encryptions $C_{ij} = (c_1, c_2)$ that hide the transferred value v_{ij} that corresponds to each cell (i, j). For instance, if we assume that there's only one receiving Bank in a transaction, the sending Bank would compute an encryption of $-v$ for its own cell, an encryption of v for the receiver cell, and a number of encryptions of 0 for the rest of the cells. This makes the transmitted values indistinguishable to any external observer due to ElGamal IND-CPA security. Additionally, the sending Bank accompanies each encryption with a NIZK proof π to prevent dishonest Bank behavior as discussed in details below. This specific ledger structure allows an external auditor to audit for a value sent/received by a Bank at any given point, with the Bank replying with a value v and a ZK proof π^{Aud} for its claim. This basic audit protocol can be extended to more complex queries (such as total assets held by a Bank or if a transaction exceeds a set limit) as we explain in Appendix A.

A straightforward implementation of such a transaction table, as done in zkLedger, leads to a ledger L that grows linearly to the number of posted transactions. This makes schemes like zkLedger hard to adopt in practice, since every single participant would have to store a table of size n times the total number of transactions that have ever occurred. Besides storage costs, the overall computational performance would also degrade even more over time.

Reducing Storage Costs. The main idea for MiniLedger, is that each Bank B_j periodically initiates a pruning operation, which prunes all transactions in its corresponding column on L. When a Bank performs a pruning operation, it publishes a digest D containing the pruned transactions and the consensus layer verifies that D is indeed a valid digest (i.e. contains the transactions to be pruned). Note B_j is still responsible for maintaining a private copy of *all* its pruned transactions, however, there are great storage savings for the public version of the ledger L which *everyone* in the system has to maintain. We note that the cost of a pruning operation depends on the number of transactions to be pruned but is independent of the number of participating Banks n.

When B_j is audited for a pruned transaction value v_{ij}, it would have to present the needed data to the auditor by recovering it from its private copy of its transaction history, and prove to the auditor that this data is contained in D and it had been posted on the specific row i (to maintain ordering). We implement this pruning operation using cryptographic accumulators to achieve a short, constant size representation of D.

4.1 Our Construction

For our construction we assume the following building blocks: the variant of the ElGamal encryption ($\mathsf{SetupTEG}, \mathsf{GenTEG}, \mathsf{EncTEG}$, DecTEG), an EU-CMA signature scheme ($\mathsf{SignGen}, \mathsf{Sign}, \mathsf{SVrfy}$), an additive, positive cryptographic accumulator ($\mathsf{AccSetup}, \mathsf{Add}$, $\mathsf{MemWitCreate}$, $\mathsf{MemWitUp}, \mathsf{VerMem}$), the Pedersen commitment scheme (ComGen, $\mathsf{Com}, \mathsf{Open}$), a consensus protocol $\mathsf{Conscus}$ and a NIZK proof system. The phases of MiniLedger work as follows:

Setup: Setup can be executed with the help of a trusted third party or via an MPC protocol amongst Banks.

1. $\mathsf{SysSetup}\{\mathsf{TP}(1^\lambda) \leftrightarrow [B_j(v_j)]_{j=1}^n\}$. This interactive protocol is executed between TP and a set of n Banks. TP verifies the initial asset value v_j for each Bank and generates the public parameters for the accumulator by running $\mathsf{AccSetup}()$, the key parameters of the ElGamal variant encryption scheme by executing $\mathsf{SetupTEG}()$ (which are also used for the Pedersen commitment scheme), the consensus protocol parameters by running $\mathsf{TPCSetup}$, and the joined set of parameters denoted as pp is sent to all Banks. Each Bank generates an ElGamal key pair ($\mathsf{pk}_{\mathsf{B_j}}, \mathsf{sk}_{\mathsf{B_j}}$) through $\mathsf{GenTEG}()$ and sends $\mathsf{pk}_{\mathsf{B_j}}$ to TP. Finally, TP encrypts the initial values of each Bank by running $C_{0j} = (c_1^{(0j)}, c_2^{(0j)}) \leftarrow \mathsf{EncTEG}(\mathsf{pk}_{\mathsf{B_j}}, v_j)$[1]. Then, it initializes a "running value total" which starts as $Q_{0j} = C_{0j}$ and will hold the encryption of the total assets of each Bank at any point. The vector $[Q_{0j}, C_{0j}]_{j=1}^n$ consists of the "genesis" state of the ledger L along with the system parameters pp containing the key parameters and all Bank public keys. At any point, the ledger L is agreed by the consensus participants and we assume that all Banks store it. pp and L are default inputs everywhere below.

Transaction Creation

2. $\mathsf{tx}_i \leftarrow \mathsf{CreateTx}(\mathsf{sk}_{\mathsf{B_k}}, [v_{ij},]_{j=1}^n)$. This algorithm is run by Bank B_k wishing to transmit some (or all) of its assets to other Banks in L. For each B_j in L (including itself), B_k executes $C_{ij} \leftarrow \mathsf{EncTEG}(\mathsf{pk}_{\mathsf{B_j}}, v_{ij})$ and computes $Q_{ij} \rightarrow Q_{(i-1)j} \cdot C_{ij}$. In order to prove *balance*, similarly to [34], B_k should pick randomness values for the ElGamal variant encryptions such that $\sum_{j=1}^n r_{ij} = 0$. Then, the sending Bank B_k generates a NIZK π_{ij} $\forall j \in (1, ..n)$ which proves the following (the exact description of π_{ij} can be found in the full version [15]):
Proof of Assets: Shows that *either* a) B_j is receiving some value ($v_{ij} \geq 0$), *or* b) B_j is spending no more than its total assets ($\sum_{k=1}^i v_{kj} \geq 0$) and within the valid range after transaction execution, while proving knowledge of its secret key $\mathsf{sk_j}$ showing it authorized the transfer. In both cases, an auxiliary

[1] To simplify notation, from now on we will drop the superscripts from the two parts of Elgamal ciphertext, i.e., we will simply write $C_{0j} = (c_1, c_2)$.

commitment $\mathsf{cm_{ij}}$ is used which commits to either v_{ij} or $\sum_{k=1}^{i} v_{kj}$, so the proof includes a single range proof for the commitment value to reduce computational costs, as the range proof is the most computationally expensive part of π.

Proof of Consistency: Ensures consistency for the encryption randomness r in c_1 and c_2 in both cases of the previous sub-proof, which guarantees correct decryption by Bank k.

The transaction $\mathsf{tx}_i = [C_{ik}, \mathsf{cm_{ik}}, \pi_{ik}, Q_{ik}]_{k=1}^{n}$ is sent to consensus layer CN.

3. $\mathsf{VerifyTx}(\mathsf{tx}_i) := b_i$. Verify all ZK proofs $[\pi_j]_{j=1}^{n}$, check that $\prod_{j=1}^{n} c_2^{(ij)} = 1$ (proof of balance) and that $Q_{ij} = Q_{(i-1)j} \cdot C_{ij}$. On successful verification output 1, else output $\perp$.

Transaction Pruning

4. $(D_{\beta j}, st'_j, \sigma_j) \leftarrow \mathsf{Prune}(st_j)$ This algorithm is executed by B_j when it wishes to prune its transaction history of depth $q = \beta - \alpha$ and "compact" it to an accumulator digest $D_{\beta j}$, where α is the latest digest and β is a currently posted row number (usually a Bank will prune everything between its last pruning and the latest transaction that appeared in L). It parses C_{ij} from each tx_{ij}. It fetches its previous digest $D_{\alpha j}$ (if $\alpha = 1$, sets $D \rightarrow D_{\alpha j}$ as the initial accumulator value where A is defined from pp). Then $\forall C_{ij}, i \in [\alpha, \beta]$ it consecutively runs accumulator addition $\mathsf{Add}(D_{(i-1)j}, (i \parallel C_{ij}))$ (note the inclusion of index i which preserves ordering of pruned transactions in D_j). Finally it stores all transaction encryptions $[i, C_{ij}]_{i=\alpha}^{\beta}$ to its local memory, updates st_j to st'_j, computes $\sigma_j \leftarrow \mathsf{Sign}(D_{\beta j})$ and sends $D_{\beta j}, \sigma_j$ to CN. Note that $D_{\beta j}$ does not include proofs π, and pruning breaks proofs of balance in rows for all Banks. Still "breaking" these old proofs is not an issue, as they have already been verified.
5. $\mathsf{PruneVrfy}(D_{\beta j}, \sigma_j) := b_j$ On receipt of $D_{\beta j}$, locally executes $\mathsf{Prune}()$ for the same transaction set to compute $D'_{\beta j}$. If $D'_{\beta j} = D_{\beta j}$ (given the accumulator is deterministic) outputs 1, else outputs $\perp$.

We note that after a *successful* pruning operation (i.e. one that is agreed upon in consensus layer), all system participants that store L can delete all existing data in cells $(i, j) \forall i < \beta$ and just store $D_{\beta j}$ along with the latest $Q_{\beta j}$.

Consensus Protocol: This is handled in the consensus layer CN with its details orthogonal to our scheme. Similar to typical blockchain consensus, participants will only update L with a new tx or D if this is valid according to the corresponding vefirication algorithms (i.e. in Bitcoin, consensus participants validate transactions before posting them in L).

6. $\mathsf{Conscus}(\mathsf{tx}$ or $D) := L'$. Runs the consensus protocol among S_{CN} to update the ledger with a new tx or pruning digest D after checking their validity. If consensus participants come to an agreement, L is updated to a new state L'.

Auditing: Our auditing protocols below include a basic audit for a value v (that has either been pruned or not) and a set's sum of such values (which might be all past transactions, thus auditing Bank's total assets). These audits are interactive and require the Bank's consent. MiniLedger can support additional audit types and/or non-interactive audits as we discuss in Appendix A.

7. $\mathsf{Audit}\{\mathsf{A}(C_{ij}) \leftrightarrow \mathsf{B}_j(\mathsf{sk_j})\}$ is an interactive protocol between an auditor A and a Bank B_j. In this basic audit, A audits B_j for the value v_{ij} of a specific transaction tx_{ij} (that has not been pruned from L so far), encrypted as C_{ij} on the ledger L. B_j first decrypts the encrypted transaction through $\mathsf{DecTEG}()$ and sends v_{ij} to A, as well as a NIZK $\pi^{\mathsf{Aud}} : \{(\mathsf{sk_j}) : c_2/g^{v_{ij}} = (c_1)^{1/\mathsf{sk_j}}\}(c_1, c_2, v_{ij}, \mathsf{pk_j}, g, h)$. Then A accepts the audit for v_{ij} if π^{Aud} successfully verifies.
8. $\mathsf{AuditSum}\{\mathsf{A}([C_{ij}]_{i=\alpha}^{\beta}) \leftrightarrow \mathsf{B}_j(\mathsf{sk_j})\}$ is an interactive protocol between an auditor A and a Bank B_j. Here A audits B_j for the sum of the values $\sum_{k=\alpha}^{\beta} v_{kj}$ for transactions $\mathsf{tx}_{\alpha j}$ up to $\mathsf{tx}_{\beta j}$ (that have not been pruned from L so far). This protocol is a generalization of the $\mathsf{Audit}\{\}$ protocol outlined above, (with $\mathsf{Audit}\{\}$ having as inputs $(\prod_{i=\alpha}^{\beta} C_{ij})$ and $\prod$ denoting direct product for ciphertexts c_1, c_2), because of ElGamal variant additive homomorphism. Note that although in this protocol the transactions are assumed to be consecutive for simplicity, its functionality is identical if the transactions are "isolated". Also if indices $\alpha = 1$ and β equals to the most recent transaction index (and no pruning has happened in the system), the audit is performed on the Bank's total assets.
9. $\mathsf{AudPruned}\{\mathsf{A}([(i,j)]_{i=\alpha}^{\gamma}, [C_{ij}]_{i=\gamma}^{\beta}) \leftrightarrow \mathsf{B}_j(\mathsf{sk_j})\}$ is an interactive protocol between an auditor A and a Bank B_j, where transactions $[\mathsf{tx}_{ij}]_{i=\alpha}^{\gamma}$ have been pruned from L (and thus the auditor only knows their indices and nothing else), and transactions $[\mathsf{tx}_{ij}]_{i=\gamma}^{\beta}$ which are still public in L (i.e. not pruned) and thus the auditor still sees their encryptions. This protocol generalizes $\mathsf{AuditSum}\{\}$. It allows the auditor to audit B_j for: (a) specific transactions (pruned or not) and, (b) sums of assets (pruned or not). For case (a), besides auditing a transaction with index in $[\gamma, \ldots, \beta]$ which is still in L, the auditor can also audit B_j for a specific transaction that has been pruned from L (i.e. ask: "Which was the value of the i-th transaction?"). The Bank would respond with the corresponding C_{ij} and depending on the underlying accumulator used, B_j would also provide a proof that C_{ij} is a member of its pruned history D_j with index i. For case (b), an auditor can audit the total (or a range of) assets of B_j no matter what transaction information of B_j remains on L. Auditing total assets works as follows: B_j fetches the stored transaction encryptions $[C_{ij}]_{i=\alpha}^{\gamma}$ from its local memory st_j, computes $[w_{ij}]_{i=\alpha}^{\gamma} \leftarrow \mathsf{MemWitCreate}(D_j, [C_{ij}]_{i=\alpha}^{\gamma}, st_j)$. Then A reads D_j from L and executes $\mathsf{VerMem}(D_j, (i \parallel C_{ij}), w_{ij}) \forall i \in (\alpha, \gamma)$, outputting $[b_{ij}]_{i=\alpha}^{\gamma}$. For every i, if $b_{ij} == 1$ it executes the $\mathsf{Audit}\{\}$ protocol with C_{ij} as input.
10. $\mathsf{AudTotal}\{\mathsf{A}(Q_{ij}) \leftrightarrow \mathsf{B}_j(\mathsf{sk_j})\}$ is equivalent to $\mathsf{Audit}\{\}$ for auditing B_j's total assets $\sum_{i=1}^{m} v_{ij}$ instead of a single v_{ij}.

Table 2. MiniLedger architecture and pruning.

(a) Ledger state before pruning, assuming B_1 had pruned before at tx_{10}.

	B_1	...	B_n
tx_1 ... tx_9	$D_{9,1}, Q_{9,1}$		...
tx_{10}	$C_{10,1} = (c_1 = \mathsf{pk}_1{}^{r_{10,1}},\ c_2 = g^{v_{10,1}} h^{r_{10,1}})$ $\pi_{10,1}, \mathsf{cm}_{10,1}, Q_{10,1}$		...
tx_{11}	$C_{11,1} = (c_1 = \mathsf{pk}_1{}^{r_{11,1}},\ c_2 = g^{v_{11,1}} h^{r_{11,1}})$ $\pi_{11,1}, \mathsf{cm}_{11,1}, Q_{11,1}$		...

(b) Ledger state after B_1 prunes at tx_{12}. Digest $D_{11,1}$ represents $C_{10,1}, C_{11,1}$ and ciphertexts that were represented in $D_{9,1}$.

	B_1	...	B_n
tx_1 ... tx_{11}	$D_{11,1}, Q_{11,1}$		...
tx_{12}	$C_{12,1} = (c_1 = \mathsf{pk}_1{}^{r_{12,1}},\ c_2 = g^{v_{12,1}} h^{r_{12,1}})$ $\pi_{12,1}, \mathsf{cm}_{12,1}, Q_{12,1}$		...

MiniLedger architecture is shown in Table 2. We informally discuss its security in Appendix A and we provide a rigorous analysis in [15].

4.2 Discussion and Comparisons

Although MiniLedger architecture resembles zkLedger [34], there exist crucial differences that make MiniLedger superior both in terms of efficiency and security. We give an overview below, and a thorough analysis in the full version [15].

Storage. As already discussed, MiniLedger by leveraging consensus properties applies a pruning strategy which achieves $O(n)$ storage requirements for L, compared to $O(mn)$ for zkLedger (where m is the total number of transactions ever happened, and is a monotonically increasing value).

Security. MiniLedger does not require any out-of-band communication, as all needed information is communicated through the ledger using encryptions. On the other hand, zkLedger assumes if a Bank is actually receiving some value in a transaction, it should be notified by the sending Bank and also learn the associated value (which was hidden in the commitment) through an out-of-band channel. zkLedger however, does not require receiving Banks to be directly informed on the randomness (i.e. commitment $\mathsf{cm}_{\mathsf{ij}}$ is never opened), since they can still answer the audits correctly using the audit tokens, provided that it knows its total assets precisely. This assumption is very strong and can potentially lead to attacks, such as the *"unknown value"* attack where a malicious sending Bank informs the receiving Bank on a wrong value (or does not inform it at all), which then prevents the receiving Bank from answering audits or even participate in the system. More importantly, with transaction values communicated out-of-bank, the randomness could be included with them as well. This would make the system trivial and defeat the purpose of most of its architecture, as the ledger would consist of just Pedersen commitments and proofs of assets. In this version the above attack would not work assuming all Banks are *always* online and verify the openings in real time, which is also a very strong assumption.

Computation. MiniLedger optimizes ZK proof computation over zkLedger by combining disjunctive proof of assets and proof of consistency into a single proof, giving an efficiency gain of roughly 10% in space and computation. Additionally, while zkLedger's computation performance degrades over time (as the monotonically-increasing ledger requires more operations to construct transactions), MiniLedger achieves steady optimal performance.

On Setup Parameters. We argue that even with the use of a TP, the trust level is rather low. The parameters of ElGamal are just random generators (similar to Pedersen commitments in zkLedger) and for certain accumulator instantiations (such as Merkle trees) there is no trapdoor behind the parameter generation. Finally, the consensus setup essentially consists of choosing trapdoorless parameters (i.e. block specifics etc.) and the set of participating parties. Thus, the only trust placed in TP is to pick a valid set of participants – something that all participants can check, exactly as in zkLedger. In comparison to zkLedger (given that ElGamal parameters are the same as Pedersen commitment parameters), the only additional setup is that of the accumulator which as discussed, for certain instantiations can be completely trapdoorless.

5 Evaluation

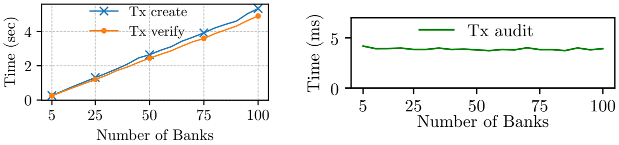
We implement a prototype of the transaction layer of MiniLedger in Python over the secp256k1 elliptic curve. We use the *zksk* library [27] for the ZK and implement range proofs using the Schoenmakers' Multi-Base Decomposition method [38][2]. The measurements were performed on Ubuntu 18.04 - i5-8500 3.0 GHz CPU - 16 GB RAM using a single thread[3].

Accumulator Instantiation
A critical implementation choice is how to instantiate the accumulator needed for the pruning operations. For efficiency reasons, we require schemes with constant size public parameters and no upper bound on the number of accumulated (i.e. pruned) elements. We only consider schemes that have at most sublinear computation and communication complexity (in the number of pruned elements) for opening/proving a single transaction to the auditor and where the auditor's verification cost is also at most sublinear.

We first consider Merkle trees [31]. Assuming a Bank prunes q transactions, the Merkle root provides $O(1)$ representation in terms of storage with $O(1)$ public parameters. Opening and verification complexity of Merkle proofs for a single transaction audit involves $O(\log q)$ hashing operations. However although hashes are relatively cheap operations, the over-linear verification complexity might be

[2] By using twisted ElGamal [17], MiniLedger is fully-compatible with Bulletproofs [11] which can further reduce its concrete storage requirements.

[3] A basic implementation of MiniLedger is available at https://github.com/PanosChtz/Miniledger.



Create	Verify	Storage	Params
53ms	49ms	68KB	2KB

Costs per Bank

Fig. 2. Transaction creation, verification and auditing costs.

a concern when auditing a series of transactions. Finally, it should be noted that Merkle trees only support membership proofs.

We then consider the batch-RSA accumulator [8]. Given that all RSA accumulators can only accumulate primes p, we use a deterministic prime mapping hash function (as in [8]) to accumulate arbitrary inputs. The batch-RSA accumulator has $O(1)$ storage for its digest with $O(1)$ public parameters as well. Proving membership for a single element p in the standard RSA accumulator, requires the prover computing a witness w equal to the primes' product in the accumulator without p (an $O(q)$ operation as shown in Fig. 4), and the verifier checking that $(g^w)^p = A$ where A is the current state of the accumulator. Therefore, batch-RSA achieves same complexity $O(q)$ for a set of elements $(p_1, ..., p_\ell)$ as when proving membership of a single element (while Merkle Trees have $O(\ell \log q)$ complexity). Consequently, the basic pruning operations $\mathsf{Prune}()$ and $\mathsf{PruneVrfy}()$ are about two orders of magnitude more expensive compared to Merkle trees as we show in Fig. 3. However they are efficient when auditing large amounts of transactions especially if auditing the total sum. Then, batching allows for negligible computation costs for the proving Bank, and negligible audit verification cost for a single transaction. Thus, choice between Merkle trees and batch-RSA accumulators ultimately relies on the use-case requirements.

For our batch-RSA implementation, we use the SHA-256 hash function and the Miller - Rabin primality test for hashing to prime numbers, and we use an RSA-3072 modulo to maintain the same level of security [35]. We decouple the witness computation cost from the proof of membership cost for the Bank, as the Bank might elect to pre-compute the witnesses before its audit (assuming however that it does not prune again until the audit, since that would require the witnesses to be recomputed again). Note the auditor needs to run the hashing to prime mapping function again for all audited values (i.e. the auditor cannot rely on the "honesty" of a Bank presenting pre-computed prime numbers for its pruned transactions). For Merkle trees, we use SHA-256 as well.

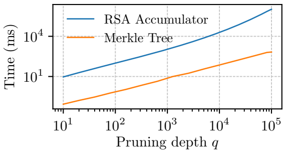


Fig. 3. Pruning computation cost

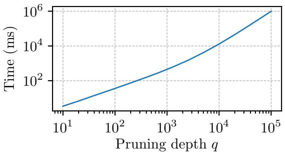


Fig. 4. RSA witness Generation cost

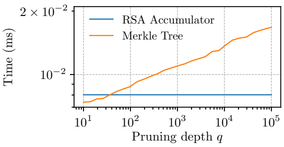


Fig. 5. Audit open cost for one tx

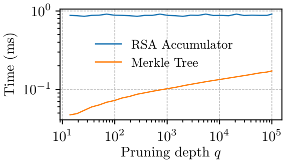


Fig. 6. Audit verify cost

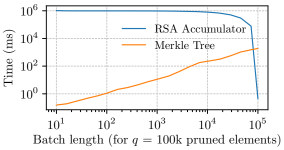


Fig. 7. Batch audit open costs

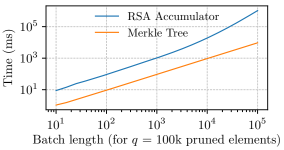


Fig. 8. Batch audit verify costs

Transaction Creation, Verification and Auditing. Every MiniLedger transaction includes an ElGamal ciphertext C, a commitment cm, a NIZK π and a running total Q for *each* Bank. Naturally, this results in linearly-increasing computation costs in terms of number of Banks as shown in Fig. 2 for both transaction creation and verification. Storing the running total Q leads to constant transaction creation and verification computational costs (for fixed number of Banks), making total assets auditing much more efficient. In contrast, zkLedger's growing ledger size also implies linearly-increasing NIZK verification costs, as the verifier would need to compute the product of all transaction elements for each Bank. The transaction creation and verification costs are 53 ms and 49 ms respectively (for a single cell in L), roughly comparable with [34].

Auditing any single value on the ledger takes about 4 ms as shown in Fig. 2. This is the cost for the complete auditing protocol, namely the decryption and proving cost for the Bank and the verification cost for the Auditor. In contrast to [34], the auditing cost is constant without being impacted either by the number of Banks or the number of past transactions.

Transaction Pruning. We evaluate the computation requirements of the pruning operation which involves executing $\mathsf{Prune}()$ and $\mathsf{PruneVrfy}()$ to create the digest D_j. Our results in Fig. 3 show it is possible to prune and verify about 1 million transactions in less than a second using Merkle trees and in about 2 h using RSA accumulator. Note prime number multiplication costs dominate the total costs (which also include hashing to primes and an exponentiation) when the pruning depth becomes large. We also stress that these computation requirements are *independent* of the number of Banks n in the system.

For transaction auditing in AudPruned{} interactive protocol, auditing opening and verification costs are shown in Figs. 5 and 6 respectively. As previously discussed, we do not include the RSA accumulator's witness creation costs (which can be pre-computed) and are shown in Fig. 4.

For auditing sums of values (i.e. "batch" auditing), the associated costs for opening and verifying a 100K transaction digest are shown in Figs. 7 and 8 respectively, with x-axis representing the number of audited transactions. Note that for auditing 10^5 transactions (i.e. the whole sum), RSA accumulator opening is significantly cheaper compared to Merkle trees, as the audited Bank would only need to retrieve the respective transactions from its local memory (which implies nearly O(1) cost) and send them to the auditor (who would in turn need to recompute all primes and perform the exponentiation of their product).

Based on our evaluation results and the discussion above, the choice between Merkle tree and batch RSA accumulator depends on use-case. Merkle trees fit a system expected to incorporate sparse audits on individual transactions, while RSA accumulator is preferred on deployments with frequent auditing on many transactions at a time (e.g. sums of assets or value thresholds over a time period).

Storage Costs. The storage cost for L has a $64n$-byte lower bound for the ElGamal variant encryptions (which represent the running total Q), plus the needed storage for each digest D and the system's pp, assuming all n Banks have pruned their transaction history and the ledger is made of a single row. Concretely, in our implementation each transaction's communication and storage cost is 68 KB per Bank (actual memory footprint), which includes the ElGamal variant encryption, the auxiliary commitment, the NIZK and the running total. A MiniLedger instantiation including the necessary public parameters, one transaction and a digest requires only 70 KB of storage per Bank.

Network and Consensus Costs. MiniLedger design focuses in the transaction layer. The consensus layer, which can be instantiated by *any* consensus protocol that satisfies the basic properties of consistency and liveness, is orthogonal to our work and providing a full implementation including a consensus layer is out of scope (note previous works [17,19,34,41] take a similar evaluation approach and do not include consensus measurements). Still for showcasing an implementation scenario, we discuss below how MiniLedger could be implemented using Hyperledger Fabric and also provide some rough cost estimations.

Table 3. Consensus costs

Banks	Peers	Tx/s	Network
10	80	21	LAN
100	4	2	WAN

For simplicity and efficiency, we consider Banks only acting as "clients", outsourcing ledger storage and consensus operations to an arbitrary number of "peer" and "orderer" nodes respectively. These numbers are entirely dependent on the use-case and does not affect MiniLedger performance or scalability. This separation between Banks and consensus participants is quite natural (e.g. Diem [2], uses similar approach i.e. decoupled permissioned consensus and provider-intermediated transactions [3]). Given Hyperledger requires at least 0.5 s to complete a full consensus operation with 4 peers and 256-bit ECDSA [4], we derive conservative estimations of the expected transaction throughput, shown in Table 3. These estimations are more than sufficient for intra-Bank transactions in a deployed system (in MiniLedger+, any number of client-to-client transactions can be aggregated in a single MiniLedger transaction). Note although permissioned consensus generally seems more fitting to MiniLedger, permissionless consensus could also be utilized when implemented on top of an smart contract.

6 Conclusion

We present MiniLedger, the first *private and auditable payment system with storage independent to the number of transactions.* MiniLedger utilizes existing cryptographic tools and innovates on the meticulous design of optimized ZK proofs to tackle important scalability issues in auditable, private payments. We achieve huge storage savings compared to previous works that store information for each transaction ever happened. Using our pruning techniques, the overall MiniLedger size can be impressively compacted to 70KB per Bank, no matter how many transactions have ever occurred. Note that our storage and computation costs could be further improved, e.g. by using Bulletproofs [11] (instead of Schoenmakers multi-base decomposition [38]), more efficient programming languages (e.g. Rust) and libraries, or by utilizing CPU parallelization. However, as in related systems [2,34] our goal is not to support "thousands" of Banks, but an arbitrary number of clients as discussed in Appendix A which does not affect the computation/storage costs in the public ledger. MiniLedger can currently serve a small consortium of Banks (e.g. the world's Central Banks) with an *arbitrary* number of clients, or build a hierarchy of a large number of Banks and clients in accordance with MiniLedger+. Evaluating MiniLedger in such a large scale or achieving its properties in a permissionless setting are interesting directions for future work.

A MiniLedger Security and Extensions

A.1 MiniLedger security

We achieve the security of MiniLedger construction as follows: *Theft prevention and balance:* relies on NIZK soundness of π (e.g. prevent a cheating prover to make false claims such as knowledge of sk or v in range) and consensus consistency. *Secure pruning:* relies on accumulator soundness (e.g. prevent accepting a digest not representing the exact set of pruned transactions) and consensus consistency. *Ledger correctness:* relies on consensus consistency. *Correct and Sound auditability:* relies on NIZK soundness (e.g. preventing convincing an auditor for a false claim), accumulator soundness and consensus consistency. *Privacy:*

relies on IND-CPA security of ElGamal variant, Pedersen commitment hiding and NIZK zero-knowledgeness (e.g. prevent distinguishing information on the ledger or leaking private information during transaction creation).

A.2 Adding Clients for Fine-grained Auditing (MiniLedger+)

At a high level, each Bank B_j maintains a *private* ledger of clients L_{B_j} (denoted as "UsrDB" in Fig. 1), independent of the public ledger L. For each client m, B_j stores its transactions in encrypted format. For a B_s client to transfer value v to a B_r client, she creates a transaction that includes encryptions of the recipient client's pk, the receiver's Bank B_r and v, as well as appropriate NIZKs to prove consistency with the protocol, which is recorded on the private ledger L_{B_s}. Then B_s constructs a transaction on L that transfers v to B_r, which in turn decrypts the information and allocates v to its recipient client. MiniLedger+ preserves anonymity while enabling fine-grained auditing at a client level, including checks that Banks allocated the funds correctly. It also has minimal overhead compared to MiniLedger while still maintaining a ledger of constant size. We provide a detailed description and analysis in the full version [15].

A.3 Additional Types of Audits

As shown in Sect. 4.1, MiniLedger basic audit functionality $\mathsf{Audit}\{\}$ is on the value v_{ij} of specific transaction tx_{ij}. Several more audit types can be constructed which reduce to that basic audit. We discuss some of those below, and provide more details for audit extensions in the full version [15]. Note these audits can still be executed for pruned data.

Full Transaction Audit: For an auditor to learn the full details of a transaction (sender, receiver and values), they would have to audit the entire row (i.e. perform n audits on v_{ij} $\forall j$).

Statistical Audits: Audits such as average or standard deviation are supported by utilizing "bit flags" to disregard zero-value transactions, proved for correctness in zero knowledge.

Value or Transactions Exceeding Limit: Utilizing appropriate range proofs, an auditor can learn if a sent or received value exceeds some limit t. Multiple range proofs can show a Bank has not exceeded the limit over a time period.

Transaction Recipient: The goal of this audit type is for a sending Bank to prove the recipients for one of its transactions. While a Bank doesn't know (and therefore cannot prove) where a *received* value came from (unless learning it out-of-band as in zkLedger), for *outbound* transactions the Bank can keep an additional record of its transaction recipients in its local memory. As an example, for proving in tx_i that the Bank really sent v_{ij} to B_j, it could send this claim to the auditor who in turn would simply then audit B_j to verify this claim.

Client Audits: Audits in a client level (e.g. statistical audits or transaction limits) can be performed similar to the respective audits in a Bank level, however the auditor needs first to learn and verify the Bank's private ledger L_B as discussed above. From that point, the auditor can perform all audits in a client level in a similar fashion to the respective audits in a Bank level. For instance, to learn if some MiniLedger+ client exceeded a value transaction threshold within a time period or over a number of transactions, this audit can be executed by selecting the client's transactions from the Bank's private table that happened within this period by their *id*'s. The audit would then be on the sum of the values represented by the product of the respective ciphertexts, and the client would produce a range proof for that ciphertext product as above. and select those with the appropriate timestamp. A special useful audit would be to learn if a MiniLedger+ client has sent assets to some specific client pk or not. The transactions would need to be augmented with an additive universal accumulator, with each sender adding the end client recipient's pk to the accumulator, while also providing its Bank a ZK proof of adding the correct public key. During an audit, the client would have to prove membership (or non membership) to the auditor. An important note is that the receiving client *does not* directly learn the original sender of a specific transaction in-band, which implies the above approach cannot work for a client to prove if he has *received* (or not) assets from another client.

Non-interactive Audits: The audit proof π^{Aud} described in Sect. 4 is interactive and require the Bank's consent. While can treat a Bank's refusal to cooperate as a failed audit, we could still enable non-interactive audits by including an encryption of π^{Aud} and its statement for each transaction cell under a pre-determined trusted auditor's public key (which preserves privacy). Our full version [15] provides more details.

References

1. Privacy coins face existential threat amid regulatory pinch. https://www.bloomberg.com/news/articles/2019-09-19/privacy-coins-face-existential-threat-amid-regulatory-crackdown
2. The libra blockchain (2020). https://developers.libra.org/docs/assets/papers/the-libra-blockchain/2020-05-26.pdf
3. Libra roles and permissions (2020). https://lip.libra.org/lip-2/
4. Androulaki, E., et al.: Hyperledger fabric: a distributed operating system for permissioned blockchains. In: Oliveira, R., Felber, P., Hu, Y.C. (eds.) Proceedings of the Thirteenth EuroSys Conference, Porto, Portugal, 23–26 April 2018, pp. 30:1–30:15. ACM (2018)
5. Androulaki, E., Karame, G.O., Roeschlin, M., Scherer, T., Capkun, S.: Evaluating user privacy in bitcoin. In: Sadeghi, A.-R. (ed.) FC 2013. LNCS, vol. 7859, pp. 34–51. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39884-1_4

6. Baldimtsi, F., et al.: Accumulators with applications to anonymity-preserving revocation. In: 2017 IEEE European Symposium on Security and Privacy, Paris, France, 26–28 April 2017, pp. 301–315. IEEE (2017)
7. Ben-Sasson, E., et al.: Zerocash: decentralized anonymous payments from bitcoin. In: 2014 IEEE Symposium on Security and Privacy, pp. 459–474. IEEE Computer Society Press (2014). https://doi.org/10.1109/SP.2014.36
8. Boneh, D., Bünz, B., Fisch, B.: Batching techniques for accumulators with applications to IOPs and stateless blockchains. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019. LNCS, vol. 11692, pp. 561–586. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-26948-7_20
9. Bonneau, J., Meckler, I., Rao, V., Shapiro, E.: Coda: decentralized cryptocurrency at scale. Cryptology ePrint Archive, Report 2020/352 (2020)
10. Bünz, B., Agrawal, S., Zamani, M., Boneh, D.: Zether: towards privacy in a smart contract world. In: Bonneau, J., Heninger, N. (eds.) FC 2020. LNCS, vol. 12059, pp. 423–443. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51280-4_23
11. Bünz, B., Bootle, J., Boneh, D., Poelstra, A., Wuille, P., Maxwell, G.: Bulletproofs: Short proofs for confidential transactions and more. In: 2018 IEEE Symposium on Security and Privacy, pp. 315–334. IEEE Computer Society Press (2018)
12. Camenisch, J., Stadler, M.: Efficient group signature schemes for large groups. In: Kaliski, B.S. (ed.) CRYPTO 1997. LNCS, vol. 1294, pp. 410–424. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0052252
13. Castro, M., Liskov, B.: Practical byzantine fault tolerance. In: Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, 22–25 February 1999, pp. 173–186 (1999)
14. Cecchetti, E., Zhang, F., Ji, Y., Kosba, A.E., Juels, A., Shi, E.: Solidus: confidential distributed ledger transactions via PVORM. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM CCS 2017, pp. 701–717. ACM Press (2017). https://doi.org/10.1145/3133956.3134010
15. Chatzigiannis, P., Baldimtsi, F.: Miniledger: compact-sized anonymous and auditable distributed payments. Cryptology ePrint Archive, Report 2021/869 (2021). https://ia.cr/2021/869
16. Chatzigiannis, P., Baldimtsi, F., Chalkias, K.: SoK: auditability and accountability in distributed payment systems. In: Sako, K., Tippenhauer, N.O. (eds.) ACNS 2021. LNCS, vol. 12727, pp. 311–337. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-78375-4_13
17. Chen, Yu., Ma, X., Tang, C., Au, M.H.: PGC: decentralized confidential payment system with auditability. In: Chen, L., Li, N., Liang, K., Schneider, S. (eds.) ESORICS 2020. LNCS, vol. 12308, pp. 591–610. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58951-6_29
18. Chepurnoy, A., Papamanthou, C., Zhang, Y.: Edrax: a cryptocurrency with stateless transaction validation. Cryptology ePrint Archive, Report 2018/968 (2018). https://eprint.iacr.org/2018/968
19. Fauzi, P., Meiklejohn, S., Mercer, R., Orlandi, C.: Quisquis: a new design for anonymous cryptocurrencies. In: Galbraith, S.D., Moriai, S. (eds.) ASIACRYPT 2019. LNCS, vol. 11921, pp. 649–678. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-34578-5_23
20. Fuchsbauer, G., Orrù, M., Seurin, Y.: Aggregate cash systems: a cryptographic investigation of mimblewimble. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019. LNCS, vol. 11476, pp. 657–689. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17653-2_22

21. Garay, J., Kiayias, A.: SoK: a consensus taxonomy in the blockchain era. In: Jarecki, S. (ed.) CT-RSA 2020. LNCS, vol. 12006, pp. 284–318. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-40186-3_13
22. Garman, C., Green, M., Miers, I.: Accountable privacy for decentralized anonymous payments. In: Grossklags, J., Preneel, B. (eds.) FC 2016. LNCS, vol. 9603, pp. 81–98. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54970-4_5
23. Gennaro, R., Jarecki, S., Krawczyk, H., Rabin, T.: Secure distributed key generation for discrete-log based cryptosystems. In: Stern, J. (ed.) EUROCRYPT 1999. LNCS, vol. 1592, pp. 295–310. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48910-X_21
24. Heasman, W.: Privacy coins in 2019: True financial freedom or a criminal's delight? (2020). https://cointelegraph.com/news/privacy-coins-in-2019-true-financial-freedom-or-a-criminals-delight
25. Jiang, Y., Li, Y., Zhu, Y.: Auditable zerocoin scheme with user awareness. In: Proceedings of the 3rd International Conference on Cryptography, Security and Privacy, Kuala Lumpur, Malaysia, 19–21 January 2019, pp. 28–32 (2019)
26. Li, Y., Yang, G., Susilo, W., Yu, Y., Au, M.H., Liu, D.: Traceable monero: anonymous cryptocurrency with enhanced accountability. IEEE Trans. Depend. Secure Comput. (2019). https://doi.org/10.1109/TDSC.2019.2910058
27. Lueks, W., Kulynych, B., Fasquelle, J., Bail-Collet, S.L., Troncoso, C.: zksk: a library for composable zero-knowledge proofs. In: Proceedings of the 18th ACM Workshop on Privacy in the Electronic Society, pp. 50–54 (2019)
28. Maxwell, G.: Confidential transactions (2015). https://people.xiph.org/~greg/confidential_values.txt
29. Maxwell, G., Poelstra, A.: Borromean ring signatures (2015). https://github.com/Blockstream/borromean_paper/blob/master/borromean_draft_0.01_34241bb.pdf
30. Meiklejohn, S., et al.: A fistful of bitcoins: characterizing payments among men with no names. Commun. ACM **59**(4), 86–93 (2016)
31. Merkle, R.C.: A digital signature based on a conventional encryption function. In: Pomerance, C. (ed.) CRYPTO 1987. LNCS, vol. 293, pp. 369–378. Springer, Heidelberg (1988). https://doi.org/10.1007/3-540-48184-2_32
32. Naganuma, K., Yoshino, M., Sato, H., Suzuki, T.: Auditable zerocoin. In: 2017 IEEE European Symposium on Security and Privacy Workshops, pp. 59–63 (2017)
33. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
34. Narula, N., Vasquez, W., Virza, M.: zkledger: privacy-preserving auditing for distributed ledgers. In: 15th USENIX Symposium on Networked Systems Design and Implementation, pp. 65–80. USENIX Association, Renton (2018)
35. National Institute of Standards and Technology: Recommendation for Key Management: NIST SP 800–57 Part 1 Rev 4. USA (2016)
36. Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 129–140. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-46766-1_9
37. Poelstra, A., Back, A., Friedenbach, M., Maxwell, G., Wuille, P.: Confidential assets. In: Zohar, A., et al. (eds.) FC 2018. LNCS, vol. 10958, pp. 43–63. Springer, Heidelberg (2019). https://doi.org/10.1007/978-3-662-58820-8_4
38. Schoenmakers, B.: Interval proofs revisited. In: Workshop on Frontiers in Electronic Elections (2005)
39. Van Saberhagen, N.: Cryptonote v 2.0 (2013). https://cryptonote.org/whitepaper.pdf

40. Wood, G.: Ethereum: a secure decentralized generalised transaction ledger (2021). https://ethereum.github.io/yellowpaper/paper.pdf, Accessed 14 Feb 2021
41. Wüst, K., Kostiainen, K., Čapkun, V., Čapkun, S.: PRCash: fast, private and regulated transactions for digital currencies. In: Goldberg, I., Moore, T. (eds.) FC 2019. LNCS, vol. 11598, pp. 158–178. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32101-7_11