Solidus: Confidential Distributed Ledger Transactions via PVORM

Ethan Cecchetti
Cornell University; IC3[†]
ethan@cs.cornell.edu

Fan Zhang
Cornell University; IC3[†]
fanz@cs.cornell.edu

Yan Ji
Cornell University; IC3[†]
jyamy42@gmail.com

Ahmed Kosba
University of Maryland; IC3[†]
akosba@cs.umd.edu

Ari Juels
Cornell Tech, Jacobs Institute; IC3[†]
juels@cornell.edu

Elaine Shi
Cornell University; IC3[†]
runting@gmail.com

[†]Initiative for CryptoCurrencies & Contracts

ABSTRACT

Blockchains and more general distributed ledgers are becoming increasingly popular as efficient, reliable, and persistent records of data and transactions. Unfortunately, they ensure reliability and correctness by making all data public, raising confidentiality concerns that eliminate many potential uses.

In this paper we present *Solidus*, a protocol for confidential transactions on public blockchains, such as those required for asset transfers with on-chain settlement. Solidus operates in a framework based on real-world financial institutions: a modest number of banks each maintain a large number of user accounts. Within this framework, Solidus hides both transaction values and the transaction graph (i.e., the identities of transacting entities) while maintaining the public verifiability that makes blockchains so appealing. To achieve strong confidentiality of this kind, we introduce the concept of a *Publicly-Verifiable Oblivious RAM Machine* (PVORM). We present a set of formal security definitions for both PVORM and Solidus and show that our constructions are secure. Finally, we implement Solidus and present a set of benchmarks indicating that the system is efficient in practice.

CCS CONCEPTS

• **Security and privacy → Domain-specific security and privacy architectures**;

Keywords: Confidential Transactions; Oblivious RAM; Blockchain

1 INTRODUCTION

Blockchain-based cryptocurrencies, such as Bitcoin, allow users to transfer value quickly and pseudonymously on a reliable distributed public ledger. This ability to manage assets privately and authoritatively in a single ledger is appealing in many settings beyond cryptocurrencies. Companies already issue shares on ledgers [27] and financial institutions are exploring ledger-based systems for instantaneous financial settlement.

For many of these companies, confidentiality is a major concern and Bitcoin-type systems are markedly insufficient. Those systems expose transaction values and the pseudonyms of transacting entities, often permitting deanonymization [45]. Concerns over this leakage are driving many financial institutions to restrict on-chain storage to transaction digests, placing details elsewhere [12, 36, 60]. Such architectures discard the key benefits of blockchains as centralized authoritative ledgers and reduce them to little more than a timestamping service.

The overall structure of current blockchains additionally misaligns with that of the modern financial system. The direct peer-to-peer transactions in Bitcoin and similar systems interfere with the customer-service role and know-your-customer regulatory requirements of financial institutions. Instead, the financial industry is exploring a model that we call *bank-intermediated* systems [36, 60]. In such systems a small number of entities—which we call *banks*—manage transactions of on-chain assets on behalf of a large number of users. For example, a handful of retail banks could use a bank-intermediated ledger to authoritatively record stock purchases by millions of customers. By design, bank-intermediated systems faithfully replicate asset flows within modern financial institutions.

While a number of bank-intermediated blockchain systems have been proposed, e.g., [1, 26, 62], these systems either do not provide inherently strong confidentiality or do so by sequestering data off-chain, preventing on-chain settlement. Coin mixes, e.g., [30, 41, 54, 61], and cryptocurrencies such as Monero [3] and Zcash [7] do improve confidentiality, but with notable limitations. Coin mixes and Monero provide only partial confidentiality, with demonstrated weaknesses [45, 48, 57]. Zero-knowledge Succinct Non-interactive ARguments of Knowledge (zk-SNARKs) [8], on which Zcash is built, provide strong confidentiality. Proof generation, however, is very expensive, requiring over a minute on a consumer machine for Zcash [7]. While this is feasible for a single client performing infrequent transactions, we show experimentally in this paper that adapting zk-SNARKs to a bank-intermediated system would be prohibitively expensive. zk-SNARKs also require an undesirable trusted setup and introduce engineering complexity and cryptographic hardness assumptions that financial institutions are reluctant to embrace [36].

CCS '17, October 30-November 3, 2017, Dallas, TX, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4946-8/17/10…$15.00
https://doi.org/10.1145/3133956.3134010

To address these concerns we present *Solidus*,[1] a system supporting strong confidentiality and high transaction rates for bank-intermediated ledgers. Solidus not only conceals transaction values, but also provides the much more technically challenging property of *transaction-graph confidentiality*.[2] This means that a transaction's sender and receiver cannot be publicly identified, even by pseudonyms. They can be identified by their respective banks, but other entities learn only the identities of the banks.

Solidus takes a fundamentally different approach to transaction-graph confidentiality than previous systems such as Zcash. As the technical cornerstone of Solidus, we introduce a new primitive called *Publicly-Verifiable Oblivious RAM Machine* or *PVORM*, an idea derived from previous work on Oblivious RAM (ORAM). In previously proposed applications, ORAM is used by a single client to outsource storage; only that client needs to verify the integrity of the ORAM. In Solidus, the ORAM stores user account balances. This means that any entity in the system must be able to verify (in zero-knowledge) that bank $\mathcal{B}$'s ORAM reflects precisely the set of valid transactions involving $\mathcal{B}$. To meet this novel requirement, a PVORM defines a set of legal application-specific operations and all updates must be accompanied by ZK proofs of correctness. Correctness includes requirements that account balances remain non-negative, that each transaction updates a single account, and so forth. We offer a formal and general definition of PVORM and describe an implementation incorporated into Solidus.

The introduction of PVORM provides several benefits to Solidus. First, a PVORM can be constructed with either zk-SNARKs or NIZK proofs based on Generalized Schnorr Proofs (GSPs) [16, 18]. GSPs are more efficient to construct than zk-SNARKs and do not require trusted setup, but are much slower to verify, so we explore both options. Second, unlike Zcash, Solidus's core data structure grows only with the number of user accounts, not the number of transactions over the system's lifetime. This property is especially important in high-throughput systems and minimizes performance penalties for injecting of "dummy" transactions to mitigate timing side-channels. Finally, Solidus maintains all balances as ciphertexts on the ledger. This approach supports direct on-chain settlement—a feature many systems, like Zcash, do not aim for. It also permits decryption of balances by authorized parties and allows users to prove their own balances if, for example, they wish to transfer funds away from unresponsive banks.

In addition to the PVORM component, we present a formal security model for Solidus as a whole in the form of an ideal functionality. This presentation may be of independent interest as a specification of the security requirements of bank-intermediated ledger systems. We prove the security of Solidus in this model.

Further, while Solidus targets a permissioned ledger model, it requires only a permissioned group; it is agnostic to the implementation of the underlying ledger, whether centralized or distributed. Therefore, we use the generic term ledger to denote a blockchain substrate that can be instantiated in a wide variety of ways.

Our contributions can be summarized as follows:

- *Bank-intermediated ledgers.* Our work on Solidus represents the first formal treatment of confidentiality on bank-intermediated ledgers—a new architecture that closely aligns with the settlement process in the modern financial system. Our work provides a formal security model that broadly captures the requirements of financial institutions migrating assets onto ledgers.
- *PVORM.* We introduce *Publicly-Verifiable Oblivous RAM Machines*, a new construction derived from ORAM and suitable for enforcing transaction-graph confidentiality in ledger systems. We offer formal definitions and efficient constructions using Generalized Schnorr Proofs.
- *Implementation and Experiments.* We report on our prototype implementation of Solidus and present results of benchmarking expreiments, demonstrating a lower bound on Solidus performence. We also provide a performance comparison with zk-SNARKs.

Our results are not just a new technical approach to transaction-graph confidentiality on ledgers. They also show the practicality of bank-intermediated ledger systems with full on-chain settlement.

2 BACKGROUND

We now review existing cryptocurrency schemes and approaches to their confidentiality. We then give some background on bank-intermediated system modeling and describe the technical building blocks used to achieve security and confidentiality in Solidus.

2.1 Existing Cryptocurrencies

Many popular cryptocurrencies are based on the same general transaction mechanism popularized by Bitcoin. Any user $\mathcal{U}$ may create an account ("address" in Bitcoin) with a public/private key pair. To transfer money, $\mathcal{U}$ creates a transaction T by signing a request to send some quantity of coins to a recipient.[3] Miners sequence transactions and directly publish T to the *blockchain*, an authoritative append-only record of transactions. Since only transactions are recorded, to determine the balance of $\mathcal{U}$, it is necessary to tally all transactions involving $\mathcal{U}$ in the entire blockchain. As a performance optimization, many entities maintain a balance summary—called an unspent transaction (UTXO) set in Bitcoin.

This setup publicizes all account balances and transaction details. The only confidentiality stems from the pseudonymity of public keys which are difficult—though far from impossible [45]—to link to real-world identities.

To conceal balances and transaction values, Maxwell proposed a scheme called Confidential Transactions (CT) [42]. CT operates in a Bitcoin-like model, but publishes only Pedersen commitments of balances. Transaction values are similarly hidden and balances are updated using a homomorphism of the commitments and proven non-negative using Generalized Schnorr Proofs (see below). Solidus uses an El-Gamal-based variant of CT to conceal transaction values.

Several decentralized cryptocurrency schemes aim to provide partial or full transaction-graph confidentiality. (See Section 8 for a brief overview.) As noted above, though, only those involving zk-SNARKs provide strong confidentiality of the type we seek for

[1]The *solidus* was a solid gold coin in the late Roman Empire.

[2]Pseudonymous cryptocurrencies such as Bitcoin are often viewed as graphs where nodes represent keys and edges transactions. The term *transaction-graph confidentiality* means concealing the graph's edges to guard against deanonymization attacks exploiting its structure [45].

[3]This is a simplification and details vary between systems. For example, a basic transaction in Bitcoin ("Pay-to-PubkeyHash"), takes a reference to the output from a previous transaction and includes a small script restricting the user of outputs and a mining fee.

Solidus. Zcash and offshoots such as Hawk [37], for example, conceal balances, transfer amounts, and the transaction graph. They do not, however, aim to align with the financial settlement system. Additionally, they require trusted setup and store authoritative state in a Merkle tree that grows linearly with the total system transaction history, drawbacks we avoid in the design of Solidus. As a basis for performance comparison, we describe and evaluate a zk-SNARK-based version of Solidus in Section 7.3.

2.2 Bank-intermediated Systems

Managing assets on ledgers is appealing to the financial industry.

The transfer of assets in financial markets today involves a laborious three-step process. *Execution* denotes a legally enforceable agreement between buyer and seller to swap assets, such as a security for cash. *Clearing* is updating a ledger to reflect the transaction results. *Settlement* denotes the exchange of assets after clearing. Multiple financial institutions typically act as intermediaries; when a customer buys a security, a broker or bank will clear and settle on her behalf via a clearinghouse.

Today, the full settlement process typically takes three days (T+3) for securities. This delay introduces systemic risk into the financial sector. Government agencies such as the Securities and Exchange Commission (SEC) are trying to reduce this delay and are looking to distributed ledgers as a long-term option. If asset titles—the authoritative record of ownership—are represented on a ledger, then trades could execute, clear, and settle nearly instantaneously.

Existing cryptocurrencies such as Bitcoin can be viewed as titles of a digital asset. Execution takes the form of digitally signed transaction requests, while clearing and settlement are simultaneously accomplished when a block containing the transaction is mined[4]

Today, however, banks intermediate most financial transactions. Even with Bitcoin, ordinary customers often defer account management to exchanges (e.g. Coinbase). Additionally, a labyrinthine set of regulations, such as Know-Your-Customer [49], favors bank-intermediated systems. Thus existing cryptocurrencies do not align well with either financial industry or ordinary customer needs.

Solidus aims to provide fast transaction settlement in a bank-intermediated ledger-based setting. As in standard cryptocurrencies, Solidus assumes that each user has a public/private key pair and digitally signs transactions. Solidus, however, conceals account balances and transaction amounts as ciphertexts. To do so and provide public verifiability at the same time, it relies on PVORM.

2.3 Oblivious RAM

As PVORM is heavily inspired by *Oblivious RAM* (ORAM), we provide some background here.

An ORAM is a cryptographic protocol that permits a client to safely store data on an untrusted server. The client maintains a map from logical memory addresses to remote physical addresses and performs reads and writes remotely. Ensuring freshness, integrity, and confidentiality of data in such a setting is straightforward using authenticated encryption and minimal local state. The key property of ORAM is *concealment of memory access patterns*; a polynomially-bounded adversarial server cannot distinguish between two identical-length sequences of client operations.

These properties provide an appealing building block for Solidus. Identifying an edge in the system's transaction graph can easily be reduced to identifying which account's balance changed with a transaction. Thus placing all balances in an ORAM immediately provides transaction graph confidentiality. Moreover, recent work has drastically improved the performance of ORAM. The most practical ORAM constructions maintain a small local cache on the client known as a *stash* and either organize the data blocks as a tree allowing logarithmic work on each access [58, 63], or write to completely randomized locations, resulting in constant-time writes but linear reads (so-called "write-only" ORAM) [11].

Unfortunately, standard ORAM is insufficient for Solidus. Because ORAM is designed for a client using an untrusted server, correctness simply means the ORAM reflects the client's updates. There is no notion of "valid" updates, let alone means for a client to prove an update's validity. In Solidus, clients (banks) must prove an application-defined notion of correctness for each update. Banks also cannot store a local stash, as we would no longer have all data on the ledger. To address these concerns we introduce PVORM—detailed in Section 4—a new construction inspired by ORAM.

2.4 Generalized Schnorr Proofs

Solidus makes intensive use of *Generalized Schnorr Proofs* (GSPs), a class of Σ-protocol for which practical honest-verifier zero-knowledge arguments (or proofs) of knowledge can be constructed.

Notation introduced in [16, 18] offers a powerful specification language for GSPs that call the PoK language. Using multiplicative group notation, let $G = \langle g \rangle$ be a cyclic group of prime order p.[5] If $x \in \mathbb{Z}_p$ and $y = g^x$, then $\mathsf{PoK}(x : y = g^x)$ represents a ZK proof of knowledge of x such that $y = g^x$ where g and y are known to the verifier. (This is the Schnorr identification protocol.)

The PoK specification language for GSPs is quite rich; it supports arbitrary numbers of variables as well as conjunctions and disjunctions among predicates. It has a set of corresponding standard tools based on the Schnorr identification protocol for efficient realization in practice when G has known order [16]. It is possible, additionally, using the Fiat-Shamir heuristic [29], to render GSPs non-interactive, i.e., to generate NIZK proofs of knowledge.

Solidus uses GSPs in a variety of ways to ensure account balances and PVORMs are properly updated and remain valid.

3 SOLIDUS OVERVIEW

Before delving into technical details, we give an overview of Solidus, including basic notation, trust assumptions, and security goals. We also give an architectural sketch. First, however, we give a concrete target application as motivation.

*Example 3.1 (*TradeWind Markets). TradeWind Markets, whose use case helped inform the design of Solidus, offers an example of how Solidus might support management of asset titles [60]. TradeWind is building an electronic communication network (ECN) for physical gold bullion to be traded using a bank-intermediated

[4]Strictly speaking, settlement involves an exchange of assets, and thus two transactions, but this issue lies outside the scope of our work.

[5]Solidus uses the group for elliptic curve secp256k1. We make this choice for performance, so despite elliptic curve groups typically using additive notation, we will use multiplicative notation for simplicity and generality.

ledger for settlement and title management. The physical bullion is managed by a custodian who is trusted to track inflows and outflows to and from a specifically designated vault. Each user has an account with a *holding bank*—generally a large commercial bank—which manages trades. A user may additionally buy gold from outside, send it to the vault, and sell it on the TradeWind ECN—requiring the custodian to create a record of the asset—or buy gold on the TradeWind ECN, remove it from the vault, and sell it elsewhere—requring the custodian to destroy the asset record.

Holdings are represented on the ledger as fractional ounces of gold held by individual users. To trade gold, a user authorizes her holding bank to transfer the gold to another user. Holding banks may also provide other services, such as holding gold as collateral against a loan. In such cases the bank may freeze assets, for example, until the loan is repaid.

As we shall show, Solidus can support the full asset lifecycle of a system like the TradeWind ECN while providing practical performance and strong confidentiality and verifiability guarantees.

3.1 Design Approach

Solidus has two important features that differ from existing ledger systems and make it more amenable to the financial industry.

The first is its *bank-intermediated* design: unlike nearly all systems proposed by the research community (see Section 8), Solidus aligns with the structure of the modern financial system. Each bank in Solidus has a set of customers or *users* who hold shares of some asset (e.g., securities, cryptocurrency, or gold) in their accounts. Specially designated entities called *asset notaries* record the injection of assets into the system, as we discuss below. Second, unlike other bank-intermediated systems, Solidus provides *strong confidentiality*. It conceals account balances and transaction details from non-transacting entities, placing them on the ledger as ciphertexts. It is for these reasons that Solidus uses PVORM. Each bank maintains its own PVORM on the ledger to record the identities and balances of its account.

Each transaction involves a sending user at a sending bank, and a receiving user at a receiving bank. When a user (sender) $\mathcal{U}_s$ signs a transaction and gives it to her (sending) bank $\mathcal{B}_s$, $\mathcal{B}_s$ first verifies the validity of the transaction—that it is correctly signed and $\mathcal{U}_s$ possesses the funds $\$v$ to be sent—then updates its PVORM to reflect the results of the transaction. The receiving bank performs a corresponding update on the receiving user's account.

The confidentiality properties of PVORM ensure that another entity can learn only the identities of the sending and receiving banks, not $\$v$ or the identities of the transacting users. Indeed, even the sending bank cannot identify the receiving user nor the receiving bank the sending user.[6] The public verifiability of PVORM ensures that any entity with access to the ledger can verify that each transaction is correctly processed by both banks.

Solidus is designed to be agnostic to the implementation of the underlying ledger. While it does require a mutually-aware group of banks and transaction validation by the ledger maintainers, those

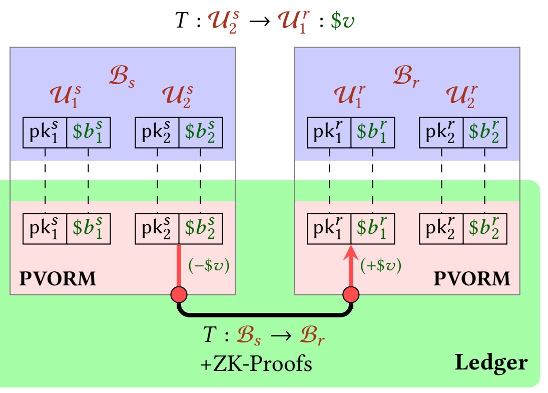
$T : \mathcal{U}_2^s \rightarrow \mathcal{U}_1^r : \$v$
$\mathcal{B}_s$
$\mathcal{U}_1^s$
$\mathcal{U}_2^s$
pk_1^s | $\$b_1^s$
pk_2^s | $\$b_2^s$
$\mathcal{B}_r$
$\mathcal{U}_1^r$
$\mathcal{U}_2^r$
pk_1^r | $\$b_1^r$
pk_2^r | $\$b_2^r$
$(-\$v)$
$(+\$v)$
PVORM
PVORM
$T : \mathcal{B}_s \rightarrow \mathcal{B}_r$
+ZK-Proofs
Ledger

Figure 1: An example transaction T where $\mathcal{U}_2^s$ at $\mathcal{B}_s$ sends $\$v$ to $\mathcal{U}_1^r$ at $\mathcal{B}_r$ and each bank has two users. The upper boxes are the logical (plaintext) memory of each bank's PVORM, and the lower boxes are the associated public (encrypted) memories. Entities other than $\mathcal{B}_s$, $\mathcal{B}_r$, $\mathcal{U}_2^s$, and $\mathcal{U}_1^r$ learn only that a user at $\mathcal{B}_s$ sent money to a user at $\mathcal{B}_r$ and both banks updated their PVORMs correctly.

maintainers can be a "permissioned" (fixed-entity) group, an "unpermissioned" (fully decentralized) ledger (a blockchain), or any other trustworthy append-only data structure.

3.2 Architectural Model

In Solidus, a predetermined set of banks $\mathcal{B}_1, \ldots, \mathcal{B}_m$ maintain asset titles on a ledger. Each bank $\mathcal{B}_i$ has a public/private key pair for each of encrypting and signing. It also has up to n users $\{\mathcal{U}_j^i\}_{j=1}^n$ each with a signature key pair. Each account is publicly associated with one bank, so $\mathrm{bank}(\mathcal{U}_j^i) = \mathcal{B}_i$ is well-defined.

Each bank $\mathcal{B}_i$ maintains its own private data structure M_i containing each user's balance and public key. It maintains a corresponding public data structure C_i, placed on the ledger, whose elements are encrypted under $\mathcal{B}_i$'s encryption key. M_i and C_i together constitute the memory in a PVORM, which we describe in Section 4. Solidus uses this structure to ensure that updates to C_i reflect valid transactions processed in M_i while concealing transaction details and the transaction graph.

A transaction T is a digitally signed request by user $\mathcal{U}_j^i$ with balance $\$b_j^i$ to send some amount $\$v$ of asset to another user $\mathcal{U}_{j'}^{i'}$. The transaction is valid if $\$b_j^i \geq \$v \geq 0$. To process a transaction, $\mathcal{B}_i$ updates M_i to set $\$b_j^i \leftarrow \$b_j^i - \$v$ and $\mathcal{B}_{i'}$ updates $M_{i'}$ to set $\$b_{j'}^{i'} \leftarrow \$b_{j'}^{i'} + \$v$. They generate publicly verifiable ZK-proofs that $\$v \geq 0$ and that they updated their respective PVORMs correctly using $\$v$. Figure 1 depicts a simple Solidus transaction.

We treat the ledger as a public append-only memory which verifies transactions. All banks have asynchronous authenticated read and write access and the ledger accepts only well-formed transactions not already present. We model this by an ideal functionality $\mathcal{F}_{\mathrm{Ledger}}$, detailed in Section 5, which any bank can invoke.

Notarizing New Asset Titles. As described above, all user transactions must be zero-sum; $\mathcal{U}_j^i$ sends money (that she must have) to $\mathcal{U}_{j'}^{i'}$. Financial systems are generally not closed, though. That is, assets can enter and leave the system through specific channels. To support this, Solidus defines a fixed set of *asset notaries*

[6] It is desirable for receiver to be able to verify the sender's identity. The sender can easily acquire a receipt by retaining a proof that she authorized the transaction.

$\{\mathcal{U}_1^\$, \ldots, \mathcal{U}_\ell^\$\}$. These are accounts with no recorded balance, but the authority to create and destroy asset titles. To ease auditing of this sensitive action, transactions involving $\mathcal{U}_i^\$$ reveal its identity.

Asset notaries clearly must be restricted; it would make no sense to allow arbitrary users to create and destroy asset titles. In Example 3.1, Solidus would designate the custodian as the sole notary responsible for acknowledging receipt and removal of the physical asset (gold) and guaranteeing its physical integrity.

3.3 Trust Model

Solidus assumes that banks respect the confidentiality of their own users but otherwise need not behave honestly. They may attempt to steal money, illicitly give money to others, manipulate account balances, falsify proofs, etc. Banks (respectively, users) can also attempt to violate the confidentiality of other banks' users (respectively, other users) passively or actively. We assume no bound on the number of corrupted banks or users, which may collude freely.

The Ledger. We assume the ledger abstraction given in Section 3.2. In practice, the ledger can, but need not, be maintained by the banks themselves. If not maintained by the banks, the ledger's trust model is independent from the higher-level protocol. It may be constructed using a (crash-tolerant) consensus protocol such as Paxos [38], ZooKeeper [32], or Raft [51], a Byzantine consensus protocol such as PBFT [20], a decentralized consensus protocol such as Nakamoto consensus [50], or even a single trustworthy entity. We simply assume that the ledger maintainers satisfy the protocol's requirements and the ledger remains correct and available.

We regard the ledger together with the public PVORM data structures $\{C_i\}$ as a replicated state machine. Despite this, Solidus's flexible design allows us to treat the consensus and application layers as entirely separate for the majority of our discussion.

Availability. We assume that the ledger remains available at all times; it is not susceptible to denial-of-service attacks and enough consensus nodes will remain non-faulty to maintain liveness. A bank, however, can be unavailable in two ways: it can freeze a user's assets by rejecting transactions or it can go offline entirely.

Asset freezing can be a feature. For certain types of assets (e.g. gold, as in Example 3.1) a user may wish to use her balance as collateral against a loan. A bank could, however, maliciously freeze a user's assets or go offline due to a technical or business failure. In either case, an auditor with the bank's decryption key (see below) could enable a user to prove her balance and recover funds despite being unable to transact directly.

Auditing. Regulators and auditors play a pivotal role in the financial sector. While Solidus does not include explicit audit support, it enables banks to prove correct decryption of on-chain data or share their private decryption key. In the first case, the auditor can acquire a transaction log on demand and verify its correctness and completeness. In the second case, the auditor can directly and proactively monitor activity within the bank and its accounts.

Network. We do not assume a network adversary. An active network adversary would make the availability requirement of the ledger impossible, while a passive adversary can be mostly mitigated simply by securing all messages with TLS. The existence of communication between users and their banks could still leak information, but this is inherent in any bank-intermediated system and could be mitigated using Tor [28] or similar protocols.

3.4 Security Goals

Solidus aims to provide very strong safety and confidentiality guarantees for both individual users and the system as a whole.

Safety Guarantees. Solidus provides a very simple but strong set of safety guarantees. First, no user's balance may decrease without explicit permission of that user (in the form of a signature), and such authorization can be used only once; there are no replay attacks. Second, account balances can never be negative, ensuring that no user can spend money she does not have. Finally, transactions that do not include asset notaries must be zero-sum.

To ensure the above properties hold, we require that the correctness of every transaction be proved in a publicly-verifiable fashion (via ZK-Proof). If the ledger checks these proofs before accepting—and settling—the transaction, then every transaction will maintain these guarantees. Solidus places all proofs on the ledger, meaning an auditor can verify them offline.

Maintaining these guarantees requires all transactions involving a single bank to be serialized. Banks can use the serialization provided by the ledger or another locking mechanism to accomplish this, but everyone must agree on the ordering.

Confidentiality Guarantees. To facilitate audits and asset recovery against malicious banks, Solidus places all account balances and transaction details directly on the ledger. Despite this persistent public record, Solidus provides a strong confidentiality for all users. First, account balances are visible only to the user's bank (and authorized auditors). Second, while transactions do reveal the sending and receiving banks, there is no way to determine if two transactions involving the same bank involved the same account. We use a hidden-public-key signature scheme (see Appendix A.3) to enforce the publicly-verifiable authorization requirement above without revealing identities. This second feature is often referred to as *transaction graph confidentiality*. It precludes use of the pseudonymous schemes employed by Bitcoin and similar systems, and is the challenge specifically addressed by PVORM.

We do not directly address information leaked by the timing of transactions. These channels and the bank-level transaction graph can, however, be eliminated by requiring each bank to post transactions at regular intervals in batches of uniform size. These batches would be padded out by "dummy" transactions of value 0 to obscure which banks conducted real transactions.

We present a formal model in Section 5 that encompasses all of these security and confidentiality goals.

4 PVORM

As discussed in Section 2.3, ORAM presents a means to conceal the Solidus transaction graph, but lacks the public verifiability that Solidus requires. To overcome this limitation, we introduce the *Publicly-Verifiable Oblivious RAM Machine* (PVORM).

As with ORAMs, PVORMs have a private logical memory M and corresponding encrypted physical memory C. There are, however, four key differences:

- *Constrained Updates.* Write operations are constrained by a public function f. In Solidus, for example, M contains account IDs and balances and f updates a single balance to a non-negative value.
- *Publicly Verifiable Updates.* Whenever the client modifies C, it must publicly prove (in zero-knowledge) that the change reflects a valid application of f.
- *Client Maintains All Memory.* Instead of a client maintaining M and a server maintaining C, the client maintains both directly. While M remains hidden, C is now publicly visible (e.g., on a ledger in Solidus).
- *No Private Stash.* Any data in M not represented in C would prevent the client from proving correctness of writes. Instead of a variable-size private stash, PVORM includes a fixed-size public encrypted stash.

To achieve public verifiability, our PVORM construction relies on public-key cryptography. Another example of an ORAM scheme that uses public key cryptography is Group ORAM [40], which does so for a more standard cloud setting, rather than our setting here. In fact, while traditional ORAMs generally uses symmetric-key primitives, this difference is not fundamental. One could construct a PVORM using symmetric-key encryption and zk-SNARKs, but as we see in Section 7.3, such a construction performs poorly.

We also leverage the fact that PVORM is designed for public verifiability and not storage outsourcing to improve efficiency. In ORAM, reads incur a cost as the client must retrieve data from the server. In PVORM, reads are "free" in that they require only reading public state—the ledger in Solidus—which leaks nothing. Writes, however, are still publicly visible. Second, since PVORM does not aim to reduce local memory usage, we assume that the client locally maintains a full copy of the PVORM including private data and metadata. This allows clients to perform updates much more efficiently by avoiding unnecessary decryption.

These features are nearly identical to those leveraged by write-only ORAM, but those techniques do not apply. Write-only ORAM requires simple writes, but we implement updates as read-update-write operations to prove properties about changes in values.

4.1 Formal Definition

We now present a formal definition of PVORM. We let M represent a private array of values from a publicly-defined space (e.g. $\mathbb{N}$) and C be the public (encrypted) representation of M. U is the space of update specifications (e.g., account ID, balance change pairs).

Definition and Correctness. We first define the public interface of a PVORM and its correct operation. A PVORM consists of the following operations.

- $\mathsf{Init}(1^\lambda, n, m_0, U) \xrightarrow{\$} (\mathsf{pk}, \mathsf{sk}, C)$, a randomized function that initializes the PVORM with security parameter 1^λ, n data elements, initial memory $M = (m_0, \ldots, m_0)$, and valid update values U.
- An update constraint function $f(u, M) \rightarrow M'$ that updates M according to update $u \in U$. Note that f may be undefined on some inputs (invalid updates), and must be undefined if $u \notin U$.
- $\mathsf{Update}(\mathsf{sk}, u, C) \xrightarrow{\$} (C', e, \mathit{proof})$, a randomized update function that takes an update u and a public memory and emits a new public memory, a ciphertext e of u, and a zero-knowledge proof of correct application.
- $\mathsf{Ver}(\mathsf{pk}, C, C', e, \mathit{proof}) \rightarrow \{\mathsf{true}, \mathsf{false}\}$, a deterministic update verification function.

We also define $\mathsf{Read}(\mathsf{sk}, C) \rightarrow M$ and $\mathsf{Dec}(\mathsf{sk}, e) \rightarrow u$, two deterministic functions that read every value from a C as a plaintext memory M and decrypt an update ciphertext, respectively. We employ these operations only in our correctness and security definitions; they are not part of the core PVORM interface.

We define correctness of a PVORM with respect to valid update sequences. An update sequence $\{u_0\}_{i=1}^k$ is *valid for* m_0 if, when $M_0 = (m_0, \ldots, m_0)$ and $M_i = f(u_i, M_{i-1})$, then M_i is defined for all $0 \le i \le k$. A PVORM is *correct* if for all initial values m_0 and all update sequences $\{u_i\}_{i=1}^k$ valid for m_0,

$$\Pr[\mathbf{Exp}^{\mathrm{Correct}}(\lambda, n, m_0, \{u_i\}_{i=1}^k)] = 1$$

where $\mathbf{Exp}^{\mathrm{Correct}}(\lambda, n, m_0, \{u_i\}_{i=1}^k)$ is defined as

```
Experiment $\mathbf{Exp}^{\mathrm{Correct}}(\lambda, n, m_0, \{u_i\}_{i=1}^k)$:
$(\mathsf{pk}, \mathsf{sk}, C_0) \overset{\$}{\leftarrow} \mathsf{Init}(1^\lambda, n, m_0, U)$
**if** $\mathsf{Read}(\mathsf{sk}, C_0) \neq M_0$, **return** false
**for** $i = 1$ to $k$ :
    $(C_i, e_i, \mathit{proof}_i) \overset{\$}{\leftarrow} \mathsf{Update}(\mathsf{sk}, u_i, C_{i-1})$
    **if** $\big[(\mathsf{Read}(\mathsf{sk}, C_i) \neq M_i) \vee (\mathsf{Dec}(\mathsf{sk}, e_i) \neq u_i)$
            $\vee \neg\mathsf{Ver}(\mathsf{pk}, C_{i-1}, C_i, e_i, \mathit{proof}_i)\big]$
        **return** false
**return** true
```

with $\{M_0, \ldots, M_k\}$ defined as above. In other words, the PVORM is correct if Update correctly transforms C as defined by f and Ver verifies these updates.

Obliviousness. Solidus requires a structure that can realize ORAM guarantees in a new setting against even an adaptive adversary. Intuitively, we require the PVORM to guarantee that any two adaptively-chosen valid update sequences result indistinguishable output. Formally, we say that a PVORM is *oblivious* if for all PPT adversaries $\mathcal{A}$, there is a negligible $\mathit{negl}(\lambda)$ such that for all $n \in \mathbb{N}$, m_0, and U,

$$\Big|\Pr\left[\mathbf{Exp}^{\mathrm{Obliv}}(0, \mathcal{A}, \lambda, n, m_0, U) = 1\right] - \Pr\left[\mathbf{Exp}^{\mathrm{Obliv}}(1, \mathcal{A}, \lambda, n, m_0, U) = 1\right]\Big| \le \mathit{negl}(\lambda)$$

where $\mathbf{Exp}^{\mathrm{Obliv}}(b, \mathcal{A}, \lambda, n, m_0, U)$ is defined by

```
Experiment $\mathbf{Exp}^{\mathrm{Obliv}}(b, \mathcal{A}, \lambda, n, m_0, U)$:
$(\mathsf{pk}, \mathsf{sk}, C) \overset{\$}{\leftarrow} \mathsf{Init}(1^\lambda, n, m_0, U)$
**return** $\mathcal{A}^{O_{b,\mathsf{sk},C}(\cdot,\cdot)}(1^\lambda, \mathsf{pk}, C)$
```

where $O_{b,\mathsf{sk},C}(\cdot,\cdot)$ is a stateful oracle with initial state $S \leftarrow C$. On input (u_0, u_1), $O_{b,\mathsf{sk},C}$ executes $(C', e, \mathit{proof}) \overset{\$}{\leftarrow} \mathsf{Update}(\mathsf{sk}, u_b, S)$, updates $S \leftarrow C'$, and returns (C', e, proof). The experiment aborts if any C' is ever undefined.

This definition is an adaptive version of those presented in the ORAM literature [56, 58, 63].

Public Verifiability. The final piece of our security definition is that of public verifiability. Intuitively, we require that each update produce a proof that the update performed was valid and is the

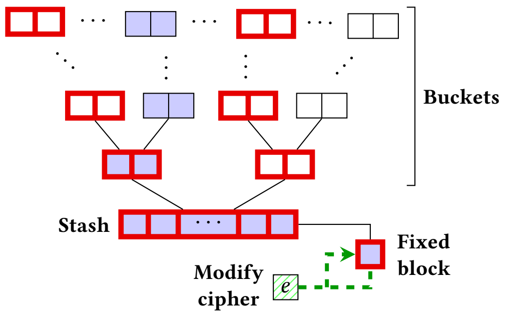

Figure 2: An update for a Circuit ORAM-based PVORM with buckets of size 2. Colors indicate the blocks involved in each operation of the read-modify-write structure. Read moves one block from the read path (shaded) into the distinguished fixed block. Then modify combines it (homomorphically) with the modify ciphertext (dashed). Finally write evicts the resulting value into the tree along two eviction paths (thick bordered).

one claimed. Formally, a PVORM is *publicly verifiable* if for all PPT adversaries $\mathcal{A}$,

$$\Pr[\mathbf{Exp}^{\mathrm{PubVer}}(\mathcal{A}, \lambda, n)] \leq \mathit{negl}(\lambda)$$

where $\mathbf{Exp}^{\mathrm{PubVer}}(\mathcal{A}, \lambda, n)$ is defined as

```
Experiment $\mathbf{Exp}^{\mathrm{PubVer}}(\mathcal{A}, \lambda, n)$:
  $(\mathsf{pk}, \mathsf{sk}, \_) \xleftarrow{\$} \mathsf{Init}(1^\lambda, n, \_, \_)$;
  $(C, C', e, \mathit{proof}) \xleftarrow{\$} \mathcal{A}(1^\lambda, n, \mathsf{pk}, \mathsf{sk})$;
  return $\mathsf{Ver}(\mathsf{pk}, C, C', e, \mathit{proof})$
    $\wedge \big(f(\mathsf{Dec}(\mathsf{sk}, e), \mathsf{Read}(\mathsf{sk}, C)) \neq \mathsf{Read}(\mathsf{sk}, C')\big)$
```

This corresponds to the soundness of the ZK-proof that an update was performed correctly.

4.2 Solidus Instantiation

In Solidus we instantiate a PVORM by combining the structure of Circuit ORAM [63] with several GSPs. Circuit ORAM places data blocks into buckets organized as a binary tree. It performs updates by swapping pairs of blocks along paths in that tree. This structure leads to good performance for two reasons: updates require logarithmic work in the number of accounts, and pairwise swaps of public-key ciphertext admit efficient ZK-proofs of correctness. Figure 2 shows how Solidus's PVORM is structured and updated.

Each data block holds an account's unique identifier and balance. This pair of values must move in tandem as blocks are shuffled, so Solidus employs a verifiable swap algorithm for El Gamal ciphertexts [34] augmented to swap ordered pairs of ciphertexts (see Appendix A.4).

Solidus constrains each update to modify one account balance and requires that balances remain in a fixed range $[0, N]$. To make updates publicly verifiable, a bank first moves the desired account to a deterministic fixed block by swapping that position with each block along the Circuit ORAM access path. Next the bank updates the account balance and generates a set inclusion proof on the resulting ciphertext to prove it is in the legal range (see Appendix A.5). Finally, the bank performs Circuit ORAM's eviction algorithm to reinsert the updated account. This again requires swapping the fixed block with a set of tree paths.

In Appendix B we concretize this construction. We prove it correct, oblivious, and publicly verifiable in the extended paper [21].

Stash Overflow. Circuit ORAM assumes a stash of bounded size, but data loss is possible if the stash overflows, resulting in a probabilistic definition of correctness; correct behavior occurs only when data is not lost. Since the probability of data loss is negligible in the size of the stash, the definition is reasonable for the setting.

In Solidus the stash must be placed on the ledger, so to prevent leaking information we also bound the stash size. Data loss is, however, catastrophic no matter how infrequent. When the stash would overflow, instead of losing data we insert one account deeper into the tree. This insertion is public, so it does leak that regular eviction was insufficient as well as the location of a single account (though not the account's identity).

Solidus inherits the stash overflow probability of Circuit ORAM, which is negligible in the stash size [63]. As we show in Section 7, the PVORM update performance is linear in the stash size, giving Solidus a direct performance-privacy trade-off. Pleasantly, modest stash sizes make overflow exceedingly unlikely. With buckets of size 3, a stash of size 25 reduces overflow probability to near 2^{-64}.

5 SOLIDUS PROTOCOL

We now present the Solidus protocol. This construction relies heavily on cryptographic primitives that we describe in Appendix A. We make this choice to simplify the explanation and leave abstract operations with several instantiations—such as range proofs.

Bank State. The state of a bank $\mathcal{B}_i$ consists of an encryption key pair $(\mathsf{ePK}_i, \mathsf{eSK}_i)$, a signing key pair $(\mathsf{sPK}_i, \mathsf{sSK}_i)$, and a set of accounts. Each account $\mathcal{U}_j$ has a unique account identifier and a balance. For simplicity, we use $\mathcal{U}_j$'s public key pk_j as its identifier.

Each bank maintains its own PVORM, updated on every transaction, containing the information of each of its accounts. Section 4.2 describes the PVORM structure.

Requesting Transactions. As Solidus is bank-intermediated, $\mathcal{U}_s$ at $\mathcal{B}_s$ must send a request to $\mathcal{B}_s$ in order to send $\$v$ to $\mathcal{U}_r$ at $\mathcal{B}_r$. The request consists of:

- A unique ID txid
- $\mathsf{Enc}(\mathsf{ePK}_s, \$v)$, $\$v$ encrypted under $\mathcal{B}_s$'s public key
- $\mathsf{Enc}(\mathsf{ePK}_r, \mathsf{pk}_r)$, a ciphertext of $\mathcal{U}_r$'s ID under $\mathcal{B}_r$'s public key
- A hidden-public-key signature signed with sk_s (see Appendix A.3).

On receipt of a request, $\mathcal{B}_s$ must validate the request—check that txid is globally unique and $0 \leq \$v \leq \$b_s$—and initiate the transaction settlement process.

Settling Transactions. Figure 3 shows the structure of settling a transaction. $\mathcal{B}_s$ generates a proof that $\$v \geq 0$, reencrypts $\$v$ under ePK_r, and sends $(\mathsf{txid}, \mathsf{Enc}(\mathsf{ePK}_r, \$v), \mathsf{Enc}(\mathsf{ePK}_r, \mathsf{pk}_r))$ to $\mathcal{B}_r$. Then both banks (concurrently) update their respective PVORMs, sign their updates, and post all associated proofs and signatures onto the ledger. Once the full transaction is accepted by the ledger, the assets have been transferred and the transaction has settled.

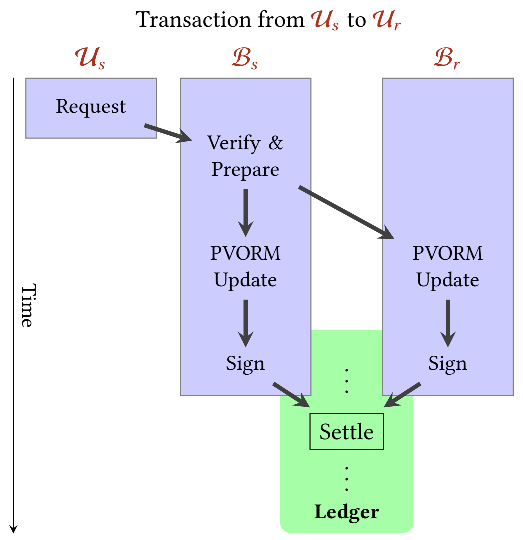

Figure 3: The lifecycle of a transaction in Solidus. An arrow from one operation to another means the second depends on the first. Note that $\mathcal{U}_r$ does not appear. The receiving user plays no role in settling transactions.

Transaction IDs. To prevent replay attacks, Solidus includes a globally unique ID with each transaction. This ID could simply be a random bit string (eg., a GUID), but then verification would require the ID of every transaction over the lifetime of the system. To avoid this growing cost, Solidus uses a two-part transaction ID: a timestamp and a random number. Transactions are only valid within a time window T_Δ. If txid $= (T, \text{id})$, the transaction is only valid at time T_{now} if $T_{\text{now}} - T_\Delta < T < T_{\text{now}}$. This allows verification to only store IDs for T_Δ and still properly prevent replay attacks.

Opening and Closing Accounts. Banks are constantly opening new accounts, so Solidus must support this. To create an account, bank $\mathcal{B}_i$ must insert the account into its PVORM. Our construction makes this simple. $\mathcal{B}_i$ publishes the new ID with a verifiable encryption of the ID and balance 0. It then inserts this ciphertext pair into its PVORM by replacing a dummy value. To close an account $\mathcal{B}_i$ simply publicly verifies the identity of an account and replaces it in the PVORM with a dummy value.

5.1 $\mathcal{F}_{\text{Ledger}}$-Hybrid Functionality

For simplicity we define the Solidus protocol, $\mathbf{Prot}_{\text{Sol}}$, using a trust initializer and an idealized ledger. We could instantiate the trusted initializer using existing PKI systems and, as mentioned above, Solidus is agnostic to the ledger implementation so we wish to leave that abstract. We present the trusted initializer $\mathcal{F}_{\text{Init}}$ in Figure 4 and the ledger $\mathcal{F}_{\text{Ledger}}$ in Figure 5. Throughout the protocol, users employ hidden-public-key signatures (see Appendix A.3) and banks employ Schnorr signatures [16, 55], denoted (sGen, Sign, sVer).

The $\mathcal{F}_{\text{Ledger}}$ functionality has two operations: posting a completed transaction and aborting an in-progress transaction. The need for the first is obvious; the ledger is the authoritative record of transactions and is responsible for their verification. The second

```
$\mathcal{F}_{\text{Init}}\left[\lambda, \{\mathcal{B}_i\}_{i=1}^{k}, \{\mathcal{U}_i\}_{i=1}^{n}\right]$
Init:
  for $i \in [1, n]$:
    Generate key pair $(\mathsf{pk}_i, \mathsf{sk}_i) \xleftarrow{\$} \mathsf{hGen}(1^\lambda)$
    send $\mathsf{pk}_i$ to each user and bank and $(\mathsf{pk}_i, \mathsf{sk}_k)$ to $\mathcal{U}_i$
  for $i \in [0, k]$:
    Generate key pair $(\mathsf{sPK}_i, \mathsf{sSK}_i) \xleftarrow{\$} \mathsf{sGen}(1^\lambda)$
    $(\mathsf{ePK}_i, \mathsf{eSK}_i, C_i) \xleftarrow{\$} \mathsf{Init}(1^\lambda, |\mathcal{B}_i|, 0, U)$
    send ("initBank", $\mathsf{ePK}_i$, $\mathsf{sPK}_i$, $C_i$) to each user and bank
    send all five values to $\mathcal{B}_i$
```

Figure 4: Solidus ideal initializer with banks $\{\mathcal{B}_i\}$ and users $\{\mathcal{U}_i\}$.

```
$\mathcal{F}_{\text{Ledger}}\left[\{\mathcal{B}_i\}_{i=1}^{k}, \{\mathcal{U}_i\}_{i=1}^{n}\right]$
Init: TXID $= \emptyset$ and LEDGER $= \epsilon$

On receive ("approveRecvTxn", txid, txn):
  assert txid $\notin$ TXID
  Parse txn $\to (\mathcal{B}_s, \mathcal{B}_r, \mathsf{txdata}_s, \sigma_s, \mathsf{txdata}_r, \sigma_r)$
  assert $\mathsf{sVer}(\mathcal{B}_s, \mathsf{txdata}_s, \sigma_s) \wedge \mathsf{sVer}(\mathcal{B}_r, \mathsf{txdata}_r, \sigma_r)$
        $\wedge\ \mathsf{VerTxn}(\mathcal{B}_s, \mathcal{B}_r, \mathsf{txn}, \mathsf{LEDGER}[\mathcal{B}_s, \mathcal{B}_r])$
  TXID $\leftarrow$ TXID $\cup$ {txid}
  LEDGER $\leftarrow$ LEDGER $\|$ (txid, txn)
  broadcast ("postTxn", txid, txn) to all banks

On receive ("abortTxn", abort) from $\mathcal{B}$:
  Parse abort $\to$ (txid, $(C, e, \mathit{proof})$, $\mathit{pf}^\star$)
  assert txid $\notin$ TXID
  assert $\mathsf{Ver}(\mathsf{ePK}, \mathsf{LEDGER}[\mathcal{B}], C, e, \mathit{proof})$
  assert $\mathit{pf}^\star$ proves $e$ is a no-op
  TXID $\leftarrow$ TXID $\cup$ {txid}
  LEDGER $\leftarrow$ LEDGER $\|$ (abort)
  broadcast ("abortTxn", abort) to all banks
```

Figure 5: Solidus ideal ledger with banks $\{\mathcal{B}_i\}$ and users $\{\mathcal{U}_i\}$. LEDGER$[\mathcal{B}_s, \mathcal{B}_r]$ denotes the most recent PVORM states for each bank in LEDGER, and VerTxn verifies all proofs associated with a given transaction, which requires the public keys and preceding PVORM state of each bank involved.

helps guard against malicious activity. As we see below, processing a transaction from $\mathcal{U}_s$ requires bank $\mathcal{B}_s$ to send its PVORM update to $\mathcal{B}_r$ prior to posting the transaction to the ledger, but $\mathcal{B}_r$ may never reply. With no abort operation, $\mathcal{B}_s$ has two options: wait for a reply—causing a DoS attack if none arrives—or proceed as if the transaction were never initiated. In the second case, $\mathcal{B}_r$ can learn with high probability whether $\mathcal{U}_s$ participates in future transactions involving $\mathcal{B}_s$; if a different Circuit ORAM path is accessed, $\mathcal{U}_s$ is not involved, but if the same path is accessed, $\mathcal{U}_s$ likely is.

In order to prevent this information leakage, $\mathcal{B}_s$ must post some PVORM update to the ledger after sending the update to $\mathcal{B}_r$ before initiating any other transaction. If the original transaction settles that includes exactly such an update. Otherwise $\mathcal{B}_s$ can invoke "abortTxn" with a dummy update on the same tree path, thus invalidating any information $\mathcal{B}_r$ may have gained.

With these ideal functionalities defined, we can now present the main Solidus protocol, $\mathbf{Prot}_{\text{Sol}}$, in Figure 6. We note that the environment $\mathcal{Z}$ is a standard UC framework entity that represents all behavior external to the protocol execution. ($\mathcal{Z}$ feeds input to and collects outputs from protocol parties and the adversary.)

```
$\mathbf{Prot}_{\mathsf{Sol}}\left[\left\{\mathcal{B}_i\right\}_{i=1}^{k}, \left\{\mathcal{U}_i\right\}_{i=1}^{n}\right]$

User $\mathcal{U}_i$:
On input ("beginTxn", $\mathcal{U}_j$, $\$v$) from environment $\mathcal{Z}$:
    Let $\mathcal{B}_i = \mathsf{bank}(\mathcal{U}_i)$ and $\mathcal{B}_j = \mathsf{bank}(\mathcal{U}_j)$
    Generate random unique txid
    Encrypt $c_r = \mathsf{Enc}(\mathsf{ePK}_j, \mathsf{pk}_j)$ and $c_v = \mathsf{Enc}(\mathsf{ePK}_i, \$v)$
    $\sigma = \mathsf{hSign}(\mathsf{sk}_i, \mathsf{ePK}_j, (\mathsf{txid}, c_r, c_v))$
    send ("requestTxn", txid, $\mathsf{ePK}_j$, $c_r$, $c_v$, $\sigma$) to $\mathcal{B}_i$

Bank $\mathcal{B}_i$:
On receive ("initBank", $\mathsf{ePK}_j$, $\mathsf{sPK}_j$, $C_j$) from $\mathcal{F}_{\mathsf{Init}}$:
    assert $B[\mathcal{B}_j]$ is not set
    $B[\mathcal{B}_j] \leftarrow (\mathsf{ePK}_j, \mathsf{sPK}_j, C_j)$

On receive ("postTxn", txid, txn) from $\mathcal{F}_{\mathsf{Ledger}}$:
    Retrieve $(\mathcal{B}_s, C'_s)$ and $(\mathcal{B}_r, C'_r)$ from txn
    if $(\mathcal{B}_i = \mathcal{B}_s$ or $\mathcal{B}_i = \mathcal{B}_r)$, then $Pend \leftarrow \bot$
    Update $B[\mathcal{B}_s] \leftarrow (\mathsf{ePK}_s, \mathsf{sPK}_s, C'_s)$
           $B[\mathcal{B}_r] \leftarrow (\mathsf{ePK}_r, \mathsf{sPK}_r, C'_r)$

On input ("abortPend") from environment $\mathcal{Z}$:
    assert $Pend \neq \bot$
    Retrieve $\mathcal{U}_i$ at $\mathcal{B}_i$ from $Pend$
    Update $(C'_i, e, \mathit{proof}) \leftarrow \mathsf{Update}(\mathsf{eSK}_i, (\mathcal{U}_i, 0), C_i)$
    Generate a ZK-proof $\mathit{pf}^\star$ that $e$ encrypts a 0-value change.
    send ("abortTxn", (txid, $\mathcal{B}_i$, $(C'_i, e, \mathit{proof})$, $\mathit{pf}^\star$) to $\mathcal{F}_{\mathsf{Ledger}}$

On receive ("abortTxn", abort) from $\mathcal{F}_{\mathsf{Ledger}}$:
    Parse (txid, $\mathcal{B}_j$, $(C'_j, e, \mathit{proof})$, $\mathit{pf}^\star$) $\leftarrow$ abort
    if $\mathcal{B}_j = \mathcal{B}_i$
        assert $Pend = (\mathsf{txid}, \_)$
        $Pend \leftarrow \bot$
    $B[\mathcal{B}_j] \leftarrow (\mathsf{ePK}_j, \mathsf{sPK}_j, C'_j)$
    if $Pend \neq \bot$ and $\mathcal{B}_j$ is the other bank in $Pend$
        Execute "abortPend" as described above

On receive ("requestTxn", txid, $\mathsf{ePK}_s$, $c_v$, $c_r$, $\sigma$) from $\mathcal{U}_s$:
    assert $(Pend = \bot) \wedge$ (txid is unique)
        $\wedge\ \mathsf{hVer}(\mathsf{ePK}_i, (\mathsf{txid}, c_v, c_r), \sigma)$
        $\wedge\ \left(((\alpha, \beta), \_) \leftarrow \sigma : \mathsf{Dec}(\mathsf{eSK}_i, (\alpha, \beta)) = \mathsf{pk}_s\right)$
    Decrypt $\$v = \mathsf{Dec}(\mathsf{eSK}_i, c_v)$
    assert $0 \leq \$v \leq M_i[\mathcal{U}_s]$
    Update $(C'_i, e_s, \mathit{proof}_s) \leftarrow \mathsf{Update}(\mathsf{eSK}_i, (\mathcal{U}_s, -\$v), C_i)$
    Let $c'_v = \mathsf{Enc}(\mathsf{ePK}_j, \$v)$
    Generate $\mathsf{txdata}_s$ containing:
      • (txid, $(c_v, c_r)$, $\sigma$, $c'_v$)
      • $(C'_i, e_s, \mathit{proof}_s)$
      • $\mathsf{RangePf}(e_v, t)$
      • Proof that $e_s$ updates $\mathcal{U}_s$ by amount in $c_v$
      • Proof that $c_v$ and $c'_v$ encrypt the same value
    $Pend \leftarrow \mathsf{txdata}_s$
    $\sigma_s = \mathsf{Sign}(\mathsf{sSK}_i, \mathsf{txdata}_s)$
    send ("approveSendTxn", txid, $\mathsf{txdata}_s$, $\sigma_s$) to $\mathcal{B}_j$

On receive ("approveSendTxn", txid, $\mathsf{txdata}_s$, $\sigma_s$) from $\mathcal{B}_j$:
    assert $(Pend = \bot) \wedge$ (txid is unique)
        $\wedge\ \mathsf{sVer}(\mathsf{sPK}_j, \mathsf{txdata}_s, \sigma_s)$
        $\wedge$ all proofs in $\mathsf{txdata}_s$ are valid
    Retrieve (txid, $(c_v, c_r)$, $\sigma$, $c'_v$) from $\mathsf{txdata}_s$
    Decrypt $\$v \leftarrow \mathsf{Dec}(\mathsf{eSK}_i, c'_v)$
    assert txid is unique and $\$v \geq 0$
    Decrypt $\mathsf{pk}_r = \mathsf{Dec}(\mathsf{eSK}_i, c_r)$
    Update $(C'_i, e_r, \mathit{proof}_r) \leftarrow \mathsf{Update}(\mathsf{eSK}_i, (\mathcal{U}_r, \$v), C_i)$
    Generate $\mathsf{txdata}_r$ containing:
      • (txid, $(c_v, c_r)$, $\sigma$, $c'_v$)
      • $(C'_i, e_r, \mathit{proof}_r)$
      • Proof that $e_r$ updates account $c_r$ by value $c'_v$
    $Pend \leftarrow (\mathsf{txid}, \mathsf{txdata}_r)$
    $\sigma_r \leftarrow \mathsf{Sign}(\mathsf{sSK}_i, \mathsf{txdata}_r)$
    Let $\mathsf{txn} = (\mathcal{B}_j, \mathcal{B}_i, \mathsf{txdata}_s, \sigma_s, \mathsf{txdata}_r, \sigma_r)$
    send ("approveRecvTxn", txid, txn) to $\mathcal{F}_{\mathsf{Ledger}}$
```

Figure 6: $\mathcal{F}_{\mathsf{Ledger}}$-hybrid protocol for Solidus with banks $\{\mathcal{B}_i\}$ and users $\{\mathcal{U}_i\}$. For simplicity we omit operations to open and close accounts.

To execute a transaction in $\mathbf{Prot}_{\mathsf{Sol}}$, a user executes "beginTxn", which sends a "requestTxn" request to the user's bank. The bank verifies the request, updates its PVORM, signs the update, and forwards it to the recipient's bank. That bank similarly verifies, updates, and signs before posting the completed transaction to $\mathcal{F}_{\mathsf{Ledger}}$. For simplicity the sending bank performs all updates and sends them to the receiving bank. In practice both banks can update their respective PVORMs in parallel as implied by Figure 3.

The protocol also contains operations for two other purposes: handling transaction aborts described above and updating other banks' states when they post updates to $\mathcal{F}_{\mathsf{Ledger}}$.

5.2 Security Definition

To demonstrate the security of $\mathbf{Prot}_{\mathsf{Sol}}$, we need a notion of how a secure Solidus protocol operates. We define this as an ideal functionality $\mathcal{F}_{\mathsf{Sol}}$ presented in Figure 7. For an adversary $\mathcal{A}$ and environment $\mathcal{Z}$, we let $\mathsf{Hybrid}_{\mathcal{A},\mathcal{Z}}(\lambda)$ denote the transcript of $\mathcal{A}$ when interacting with $\mathbf{Prot}_{\mathsf{Sol}}$. We let $\mathsf{Ideal}_{\mathcal{S},\mathcal{Z}}(\lambda)$ be the transcript produced by a simulator $\mathcal{S}$ when run in the ideal world with $\mathcal{F}_{\mathsf{Sol}}$. This allows us to define security as follows.

Definition 5.1. We say that Solidus *securely emulates* $\mathcal{F}_{Sol}$ if for all real-world PPT adversaries $\mathcal{A}$ and all environments $\mathcal{Z}$, there exists a simulator $\mathcal{S}$ such that for all PPT distinguishers $\mathcal{D}$,

$$\Big|\Pr\left[\mathcal{D}\left(\mathsf{Hybrid}_{\mathcal{A},\mathcal{Z}}(\lambda)\right) = 1\right] - \Pr\left[\mathcal{D}\left(\mathsf{Ideal}_{\mathcal{S},\mathcal{Z}}(\lambda)\right) = 1\right]\Big| \leq \mathit{negl}(\lambda).$$

This definition leads to the following theorem, which we prove in the extended version of the paper [21].

Theorem 5.2. *The Solidus protocol* $\mathbf{Prot}_{Sol}$ *satisfies Definition 5.1 assuming a DDH-hard group in the ROM.*

In order to prove Theorem 5.2 in the Universal Composability (UC) framework [19], we assume Solidus employs only universally composable (UC) NIZKs. Prior work [6] demonstrates that GSPs can be transformed into UC-NIZKs by using the Fiat-Shamir heuristic and including a ciphertext of the witness under a public key provided by a common initializer. As Solidus already employs this trusted initialization and includes ciphertexts of most operations anyway, the performance impact of ensuring UC-NIZKs is minimal.

6 OPTIMIZATIONS

In addition parallelizing operation, there are several optimizations which make Solidus more practical. Some of these optimizations are only appropriate for certain use cases, but they may result

```
$\mathcal{F}_{\text{Sol}}\left[\{\mathcal{B}_i\}_{i=1}^{k}, \{\mathcal{U}_i\}_{i=1}^{n}, \{\mathcal{U}_i^{\$}\}_{i=1}^{\ell}\right]$
Init
  Initialize $T$ to empty
  Initialize $V[\mathcal{U}_i] \leftarrow 0$ for $i \in [1, n]$

On receive ("requestTxn", $\mathcal{U}_r$, $\$v$) from $\mathcal{U}_s$:
  assert $\$v \geq 0$
  Generate unique txid
  $T$[txid] $\leftarrow (\mathcal{U}_s, \mathcal{U}_r, \$v,$ "req")
  send txid to $\mathcal{U}_s$
  send ("req", txid, $\mathcal{U}_s$, bank($\mathcal{U}_r$), $\$v$) to bank($\mathcal{U}_s$)

On receive ("approveSendTxn", txid) from $\mathcal{B}_s$:
  Retrieve $(\mathcal{U}_s, \mathcal{U}_r, \$v, f) \leftarrow T$[txid]
  assert $f$ = "req" and $\mathcal{B}_s$ = bank($\mathcal{U}_s$)
  $T$[txid] $\leftarrow (\mathcal{U}_s, \mathcal{U}_r, \$v,$ "aprv")
  send ("aprv", txid, $\mathcal{B}_s$, $\mathcal{U}_r$, $\$v$) to bank($\mathcal{U}_r$)

On receive ("approveRecvTxn", txid) from $\mathcal{B}_r$:
  Retrieve $(\mathcal{U}_s, \mathcal{U}_r, \$v, f) \leftarrow T$[txid]
  assert $f$ = "aprv" and $\mathcal{B}_r$ = bank($\mathcal{U}_r$)
  Remove $T$[txid] mapping
  Retrieve $\$b_s \leftarrow V[\mathcal{U}_s]$, $\$b_r \leftarrow V[\mathcal{U}_r]$
  assert $\$b_s \geq \$v$ or $\mathcal{U}_s = \mathcal{U}_i^{\$}$ for some $i$
  $V[\mathcal{U}_s] \leftarrow \$b_s - \$v$
  $V[\mathcal{U}_r] \leftarrow \$b_r + \$v$
  // Reveal identities of asset notaries and banks
  Let $\mathcal{P}_s = \mathcal{U}_s$ if $\mathcal{U}_s = \mathcal{U}_i^{\$}$, bank($\mathcal{U}_s$) otherwise
  Let $\mathcal{P}_r = \mathcal{U}_r$ if $\mathcal{U}_r = \mathcal{U}_j^{\$}$, bank($\mathcal{U}_r$) otherwise
  broadcast ("postTxn", txid, $\mathcal{P}_s \rightarrow \mathcal{P}_r$) to all banks

On receive ("abortTxn", txid) from $\mathcal{B}$:
  if txid has been seen before // Can "abort" nonexistent transactions
    Retrieve $(\mathcal{U}_s, \mathcal{U}_r, \_, \_) \leftarrow T$[txid]
    assert $\mathcal{B}$ = bank($\mathcal{U}_s$) or $\mathcal{B}$ = bank($\mathcal{U}_r$)
    Remove $T$[txid] mapping
  broadcast ("abortTxn", txid, $\mathcal{B}$) to all banks
```

Figure 7: Ideal functionality for the Solidus system with banks $\{\mathcal{B}_i\}$, users $\{\mathcal{U}_i\}$, and asset notaries $\{\mathcal{U}_i^{\$}\}$. For simplicity we assume a fixed set of accounts for each bank.

in significant speedups when applicable. We include the simpler optimizations in our evaluation in Section 7.

6.1 Precomputing Randomization Factors

A large computational expense in Solidus is re-randomizing ciphertexts while updating a PVORM. Fortunately, El Gamal allows us to re-randomize ciphertexts by combining them with fresh encryptions of the group identity. That is, in a group $G = \langle g \rangle$ with key pair $(\text{pk} = g^{\text{sk}}, \text{sk})$ and a ciphertext $c = (\alpha, \beta)$, we can re-randomize c by picking a random $r \leftarrow \mathbb{Z}_{|G|}$ and computing $c' = (\alpha \cdot \text{pk}^r, \beta \cdot g^r)$.

Computing (pk^r, g^r) only requires knowledge of the group G, the generator g, and a bank's public key pk, none of which change. This means we can precompute these unit ciphertexts and re-randomize by multiplying in a precomputed value.

Since the system can continue indefinitely, it must continue generating these randomization factors. Many financial systems have predictable high and low load times (e.g., very light traffic at night), so they can utilize otherwise-idle hardware for this purpose during low-traffic times. If the precomputation generates more randomization pairs than the application consumes over a modest time frame (e.g. a day), we can drastically improve performance.

6.2 Reducing Verification Overhead

As we see in Section 7, proof verification is quite expensive. In the basic protocol, the ledger consensus nodes must each verify every transaction. As more banks join the system this increases the load on the consensus nodes—which may be the banks. By strengthening trust assumptions slightly, we can omit much of this online verification and increase performance. We present two strategies that rely on different assumptions.

Threshold Verification. In the financial industry, there is often a group of entities (e.g., large banks and regulators) who are generally trusted. If a threshold number of these entities verify a transaction, this could give all other consensus nodes—often other banks—confidence that the transaction is valid, allowing them to accept it without further verification. Once the threshold is reached, each other node need only verify the signatures of the trusted entities that verified the transaction, which is far faster than performing a full verification. If the group of trusted entities is significantly larger than the threshold or those entities have much more capacity than others, this strategy will improve system scaling.

Full Offline Verification. In some cases banks can be treated as covert adversaries. That is, they will attempt to learn extra information, but they will subvert the protocol only if attribution is impossible. This situation could arise if each Solidus bank is controlled by a large commercial bank. While a bank may wish to learn as much information as possible, the cost of being caught misbehaving is high enough to deter attributable protocol deviations.

Under these assumptions we can omit online verification entirely. The verifiability of a ledger-based system remains in place, so if a bank submits an invalid transaction or proof, post hoc identification of the faulty transaction and offending bank is trivial. Thus, in this covert adversary model, banks will only submit valid transactions and proofs, meaning that the ledger can accept transactions without first verifying the associated proofs first.

6.3 Transaction Pipelining

Solidus requires sequential processing of transactions at a single bank because PVORM updates must be sequential to generate valid proofs. Given transactions T_1 followed by T_2, in order for $\mathcal{B}$ to process T_2 it needs the PVORM state following T_1. It does not, however, need the associated proofs. Therefore, if $\mathcal{B}$ assumes T_1 will settle—because faults are rare—it can start processing T_2 early while generating proofs for T_1. While this technique will not reduce transaction latency, it can drastically increase throughput. Moreover, determining the updated PVORM state requires primarily re-randomizing ciphertexts, making this optimization particularly effective when combined with precomputation (Section 6.1).

When failures do occur, it impacts performance but not correctness. If T_1 aborts for any reason, T_2 will not yet have settled since T_1 would have to settle first. This means $\mathcal{B}$ can immediately identify the problem and reprocess T_2—and any following transactions—without T_1. This reprocessing may lead to significant, but temporary, performance degradation, meaning this optimization is only appropriate when failure are rare if each transaction is posted individually to the ledger.

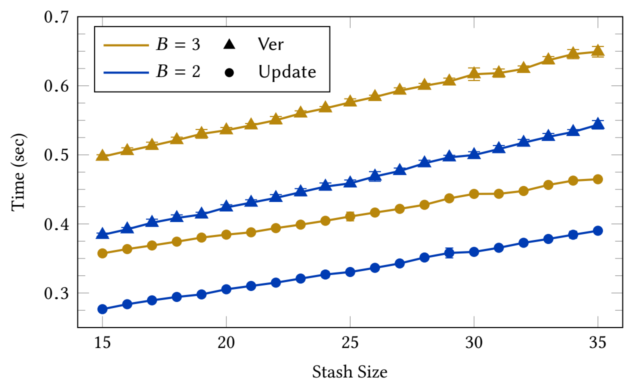

Figure 8: PVORM performance with capacity 2^{15} for buckets of size $B = 2$ and $B = 3$ as stash size varies.

We can alleviate some of this performance penalty by bundling transactions into blocks, as is done in systems like Bitcoin. If T_1 aborts, instead of reprocessing T_2, $\mathcal{B}$ can include a rollback operation later in the same block. This rollback must provably revert any changes executed by T_1's update, thus allowing verifiers to check that T_1 was never processed at all. There is, however, no need to recompute T_2 as long as the rollback can be placed after it while remaining in the same block as T_1.

7 EXPERIMENTS

We now present performance results for our PVORM and Solidus prototypes. Our Solidus implementation is 4300 lines of Java code, 2000 of which constitute the PVORM. We use BouncyCastle [14] for crypto primitives and Apache ZooKeeper [5] for distributed consensus. We ran all experiments on `c4.8xlarge` Amazon EC2 instances and employed the precomputation optimization (Section 6.1). These benchmarks do not include the precomputation time.

We emphasize that our performance results employ an unoptimized implementation and only *one server per bank*, highly limiting our parallelism. Solidus is designed to be *highly parallelized*, allowing it to scale up using multiple servers per bank to achieve *vastly superior performance* in practice.

7.1 PVORM Performance

We measured the concrete performance of PVORM Update and Ver operations under different configurations and levels of parallelism.

Bucket and Stash Size. Figure 8 shows the single-threaded performance of our PVORM as we vary bucket and stash sizes. As expected, larger buckets are slower and runtime grows linearly with the stash size. As bucket and stash sizes determine the chance of stash overflow, this measures the performance-privacy trade-off.

Tree Depth. Figure 9 shows the single-threaded performance of our PVORM as the capacity scales. As expected, the binary tree structure results in clearly logarithmic scaling.

Parallelism. Our PVORM construction supports highly parallel operation. A single update contains many NIZKs that can be created or verified independently. Figure 10 shows performance for a single

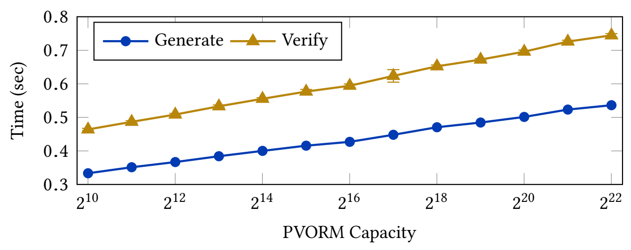

Figure 9: PVORM capacity scaling with buckets of size 3 and stash of size 25.

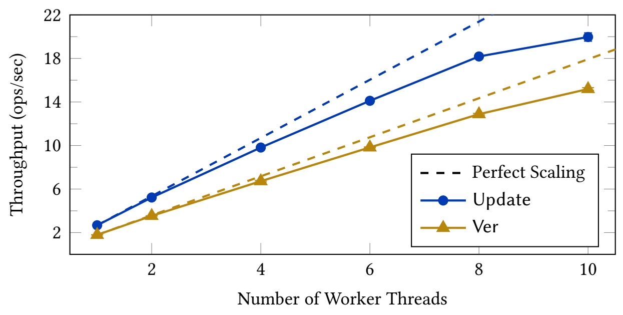

Figure 10: Parallel PVORM performance using size 3 buckets, a size 25 stash, and capacity of 2^{15}. Dashed lines show perfect scaling where all computation is parallelized with no overhead.

PVORM with varying numbers of worker threads. In each test there is exactly one coordination thread, which does very little work.

Because the proof of each pairwise swap can be computed or verified independently, we expect performance to scale well beyond 10 threads—possibly as high as 100. We stop at 10 for a combination of two reasons. First, PVORM operations are CPU-bound, so adding threads beyond the number of CPU cores produces no meaningful speedup. Second, our prototype implementation does not distribute to multiple hosts and scales poorly to multi-CPU architectures. Since `c4.8xlarge` EC2 instances have two 10-core CPUs, we present scaling to only 10 worker threads. Note that with 10 worker threads there are 11 total threads, so some work may not be effectively parallelized on the same CPU. This likely explains some of the reduced scaling in that case.

Proof Size and Memory Usage. For a PVORM with size 3 buckets, a size 25 stash, and capacity 2^{15}, a single PVORM update with proofs is 190 KB (or 114 KB if compressed[7]). To generate an update, our prototype requires a complete copy of the PVORM in memory. Despite this, memory consumption peaks at only 880 MB.

7.2 Solidus System Performance

We present performance tests of a fully distributed Solidus system with 2 to 12 banks. Each bank runs on its own `c4.8xlarge` EC2 instance and maintains a PVORM with size 3 buckets, as size 25 stash, and capacity 2^{15}. These parameters allow realistic testing, with a stash overflow probability of around 2^{-64}. To maintain the ledger,

[7] An elliptic curve point is an ordered pair of elements of $\mathbb{F}_p$. Points can be compressed to a single bit and a field element, but decompression imposes nontrivial overhead.

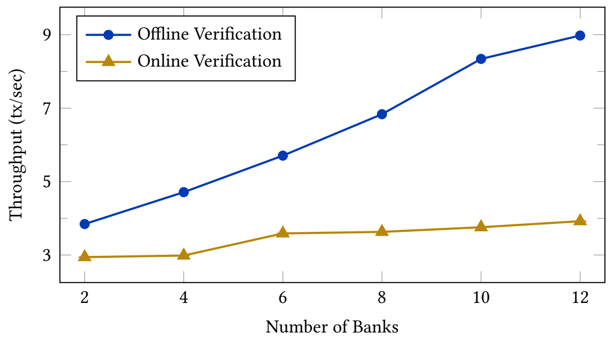

Figure 11: Solidus performance distributed using ZooKeeper. Each bank is a ZooKeeper node and maintains a PVORM with size 3 buckets, a size 25 stash, and capacity 2^{15}.

each bank's host also runs a ZooKeeper [5] node. We make no attempt to tune ZooKeeper or optimize off-ledger communication.

To test this configuration we fully load each bank with both incoming and outgoing transactions. As explained in Section 6.2, in some settings transaction verification can occur offline, so we also test performance with online verification turned off.

Figure 11 contains the results of these tests. With regular online verification, performance improves until all CPUs are saturated verifying third-party transactions, after which point scaling slows. Using offline verification, transactions settle faster and additional banks impose lower overhead on existing banks, improving scaling.

These results could be further improved by having each bank distribute verification cross multiple machines, improving capacity and throughput. Pipelining transactions (as described in Section 6.3) could improve throughput substantially if banks also distributed proof generation across multiple hosts. (Such distribution is unlikely to provide any benefit without pipelining.) Implementing this distribution introduces complex systems engineering challenges that are orthogonal to the technical innovations introduced by Solidus, so we neither implement nor benchmark these options.

7.3 zk-SNARK Comparison

We finally compare our prototype's performance to that of a PVORM implemented with zk-SNARKs. This approach has succinct proofs and short verification times, but costly proof generation.

Simply taking our Circuit ORAM PVORM construction and converting all proofs to zk-SNARKs would be needlessly expensive. As zk-SNARKs can prove correct application of an arbitrary circuit [8], we use a compact Merkle tree structure. Each account is stored at the leave of a standard Merkle hash tree, the root of which is posted to the ledger. To update the PVORM, a bank updates one account to a valid value and modifies the Merkle tree accordingly. It then produces a zk-SNARK that it properly performed the update and verified the requester's signature. The root of the new Merkle tree is the new PVORM state and the zk-SNARK is the proof.

We implemented this construction using a security level equivalent to our GSP-based PVORM.[8] Table 1 shows its performance

[8]Both hash with SHA-256. The GSP-based PVORM uses El Gamal with the secp256k1 curve and the SNARK-based PVORM uses RSA-3072. Both provide 128 bits of security.

	Number of Threads		
	1	4	36
Proof Time (sec)	65.45	24.53	13.76
Verification Time	0.0065 sec		
Proof Size	288 bytes		
Peak Memory Use	7.2 GB		

Table 1: Performance of PVORM using zk-SNARKs.

running on a `c4.8xlarge` EC2 instance. While verification is extremely fast, even highly parallel proof generation is more than 200 times slower than the GSP PVORM. For this to improve overall system throughput, the system would need to verify every proof around 200 times. In our expected use-case, at most tens of banks would maintain the ledger, so this is significantly slower. Moreover, additional hardware can allow banks to verify numerous GSP transactions in parallel but provides little benefit to zk-SNARKs.

8 RELATED WORK

We compare Solidus here with related work on cryptocurrencies and transaction confidentiality. We omit related work on ORAM, which was covered in Sections 2.3 and 4.

Anonymous cryptocurrencies. Anonymous e-cash was proposed by Chaum [22, 23] and refined in a long series of works, e.g., [15, 17, 31]. In these schemes, trust is centralized. A single authority issues and redeems coins that are anonymized using blind signatures or credentials. Due to its centralization and technical limitations, such as poor handling of fractional coins and double-spending, e-cash has been largely displaced by decentralized cryptocurrencies.

Zcash, a recently deployed decentralized cryptocurrency, and its academic antecedents [7, 25, 46] and offshoots e.g., Hawk [37], provide strong transaction-graph confidentiality like Solidus. Zcash relies on zk-SNARKs to ensure conservation of money, prevent double spending, and hide both transaction values and the system's transaction graph. Consequently, unlike Solidus, it requires trusted setup, which in practice must be centralized (as multiparty computation for this purpose [9] is impractical). Moreover, as we showed in our exploration of a zk-SNARK variant of Solidus in Section 7.3, zk-SNARKs are far more expensive to generate (by two orders of magnitude) than the GSPs used in Solidus. Additionally, Zcash and Hawk do not provide auditability as Solidus does; as designed, they do not record assets on-chain, only commitments.

Alternative schemes such as Monero [3], a relatively popular cryptocurrency, and MimbleWimble [35], a pseudonymous proposal, provide partial transaction-graph concealment. Serious weaknesses in Monero's anonymity have recently been identified, however [47], while MimbleWimble has yet to be deployed or have its confidentiality properties formally analyzed.

Mixes. Mixes partially obscure the transaction graph in an existing cryptocurrency. A number have been proposed and deployed, e.g., [30, 41, 54, 61]. Mixes have a fundamental limitation: they only protect participating users, and thus provide only partial anonymity, resulting in demonstrated weaknesses [48, 57]. As mixes' costs are linear in the number of participants, they do not scale well. In contrast, Solidus achieves strong and rigorously provable transaction-graph confidentiality for all users.

Confidential Transactions. A class of schemes called Confidential Transactions [39, 42, 43] hide transaction amounts, but do not aim at transaction graph privacy. Solidus employs a Confidential Transaction scheme similar to that in [42], but makes more direct use of and inherits the provable security properties of GSPs.

Financial sector blockchain technologies. The financial industry's intense interest in blockchains has led to a number of proposed and deployed systems. These systems support current banking system transaction flows like Solidus. They achieve elements of Solidus, but lack its full set of features. For example, Ripple [2] is a widely deployed scheme for value transfer, but does not aim at the confidentiality properties of Solidus. RSCoin [26], a scheme for central bank cryptocurrency issuance that supports auditability like Solidus, but similarly does not inherently support transaction confidentiality. Other examples are SETLcoin [62], which aims at on-chain trade settlement, like Solidus, but lacks strong transaction-graph confidentiality, and the Digital Asset Platform [1], which provides confidentiality by keeping transaction details off-chain and completely foregoing on-chain settlement and auditability.

9 CONCLUSION

We have introduced Solidus, a system that addresses a major impediment to broad use of blockchain transaction systems, their critical lack of *transaction-graph confidentiality*. Unlike previous approaches (e.g. Zcash), Solidus is specifically geared towards the structural and performance requirements of modern financial transaction and settlement systems. The key innovation in Solidus is the Publicly-Verifiable Oblivious RAM Machine (PVORM), a generalization of ORAM. A PVORM supports publicly verifiable outsourcing of computation over memory, enabling a completely new approach to blockchain transaction system design. Solidus employs a PVORM with data structure size linear in the number of accounts—rather than the number of transactions in the system, as in Zcash—and proof computation times two orders of magnitude faster than zk-SNARKs. We define the security of Solidus as an ideal functionality and prove its security in the UC framework. Finally, we present a series of optimizations and experiments running the complete Solidus protocol on a distributed ledger (ZooKeeper), which demonstrate the ability of Solidus to scale to the throughputs required for real-world workloads. We believe that Solidus is the first viable approach to building strongly verifiable and fully auditable bank-intermediated ledger transaction systems.

ACKNOWLEDGEMENTS

This work is funded in part by NSF grants CNS-1314857, CNS-1330599, CNS-1453634, CNS-1518765, CNS-1514261, CNS-1514163, and CNS-1564102, ARO grant W911NF-16-1-0145, a Packard Fellowship, a Sloan Fellowship, Google Faculty Research Awards, a VMWare Research Award, and IC3 sponsorship from Chain, IBM, and Intel. We would like to thank Matthew Trudeau and Abishek Kumarasubramanian at TradeWind Markets and Shaul Kfir and Tamás Blummer at Digital Asset Holdings for patiently explaining the needs, requirements, and background of the financial industry. We would also like to thank Eleanor Birrell, Philip Daian, Joshua Gancher, Andrew Morgan, Isaac Sheff, and the anonymous reviewers for their insightful comments and help editing.

REFERENCES

[1] 2017. Digital Asset Plaform. www.digitalasset.com. (2017).
[2] 2017. Ripple. www.ripple.com. (2017).
[3] Referenced May 2017. Monero. www.getmonero.org. (Referenced May 2017).
[4] Ittai Anati, Shay Gueron, Simon P Johnson, and Vincent R Scarlata. 2013. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy.*
[5] Apache Software Foundation. 2016. Apache ZooKeeper (Version 3.4.9). https://zookeeper.apache.org/. (2016).
[6] Boaz Barak, Ran Canetti, Jesper Buus Nielsen, and Rafael Pass. 2004. Universally Composable Protocols with Relaxed Set-up Assumptions. In *FOCS.*
[7] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *IEEE Symposium on Security and Privacy.*
[8] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. 2013. SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge. In *CRYPTO.*
[9] Eli Ben-Sasson, Alessandro Chiesa, Matthew Green, Eran Tromer, and Madars Virza. 2015. Secure Sampling of Public Parameters for Succinct Zero Knowledge Proofs. In *IEEE Symposium on Security and Privacy.* 18.
[10] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture. In *USENIX Security.*
[11] Erik-Oliver Blass, Travis Mayberry, Guevara Noubir, and Kaan Onarlioglu. 2014. Toward Robust Hidden Volumes Using Write-Only Oblivious RAM. In *CCS.*
[12] Tamás Blummer. 2016. Personal communication with Tamás Blummer, Chief Ledger Architect, Digital Asset Holdings. (2016).
[13] Fabrice Boudot. 2000. Efficient proofs that a committed number lies in an interval. In *EUROCRYPT.*
[14] BouncyCastle. 2016. Bouncy Castle Crypto APIs (Version 1.55). https://www.bouncycastle.org/. (2016).
[15] Jan Camenisch, Susan Hohenberger, and Anna Lysyanskaya. 2005. Compact e-cash. In *EUROCRYPT.*
[16] Jan Camenisch, Aggelos Kiayias, and Moti Yung. 2009. On the Portability of Generalized Schnorr Proofs. In *EUROCRYPT.*
[17] Jan Camenisch, Anna Lysyanskaya, and Mira Meyerovich. 2007. Endorsed E-Cash. In *IEEE Symposium on Security and Privacy.*
[18] Jan Camenisch and Markus Stadler. 1997. Efficient group signature schemes for large groups. In *CRYPTO.*
[19] Ran Canetti. 2001. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS.*
[20] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *OSDI.*
[21] Ethan Cecchetti, Fan Zhang, Yan Ji, Ahmed Kosba, Ari Jules, and Elaine Shi. 2017. Solidus: Confidential Distributed Ledger Transactions via PVORM, Extended Version. Cryptology ePrint Archive, Report 2017/317. (2017). https://eprint.iacr.org/2017/317.
[22] David Chaum. 1982. Blind signatures for untraceable payments. In *CRYPTO.*
[23] David Chaum, Amos Fiat, and Moni Naor. 1990. Untraceable electronic cash. In *CRYPTO.*
[24] Ivan Damgård. 2002. On Σ-protocols. *Lecture Notes, University of Aarhus, Department for Computer Science* (2002).
[25] George Danezis, Cedric Fournet, Markulf Kohlweiss, and Bryan Parno. 2013. Pinocchio Coin: building Zerocoin from a succinct pairing-based proof system. In *PETShop.*
[26] George Danezis and Sarah Meiklejohn. 2016. Centrally Banked Cryptocurrencies. In *NDSS.*
[27] Michael del Castillo. 16 December 2016. Overstock Just Closed its First Day of Blockchain Stock Trading. *Coindesk* (16 December 2016).
[28] Roger Dingledine, Nick Mathewson, and Paul Syverson. 2004. Tor: the second-generation onion router. In *USENIX Security.*
[29] Amos Fiat and Adi Shamir. 1986. How to prove yourself: Practical solutions to identification and signature problems. In *EUROCRYPT.*

[30] Ethan Heilman, Leen AlShenibr, Foteini Baldimtsi, Alessandra Scafuro, and Sharon Goldberg. 2017. TumbleBit: An untrusted Bitcoin-compatible anonymous payment hub. In *NDSS*.

[31] Gesine Hinterwälder, Christian T. Zenger, Foteini Baldimtsi, Anna Lysyanskaya, Christof Paar, and Wayne P. Burleson. 2013. Efficient E-Cash in Practice: NFC-Based Payments for Public Transportation Systems. In *PETS*.

[32] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems.. In *USENIX ATC*.

[33] Intel Corporation 2016. *Intel® Software Guard Extensions SDK*. Intel Corporation. [Online; accessed 6-February-2017].

[34] Markus Jakobsson and Ari Juels. 1999. *Millimix: Mixing in small batches*. Technical Report. DIMACS Technical report 99-33.

[35] Tom Elvis Jedusor. 19 July 2016. MimbleWimble. Referenced 2017 at https://download.wpsoftware.net/bitcoin/wizardry/mimblewimble.txt. (19 July 2016).

[36] Shaul Kfir. 2016. Personal communication with Shaul Kfir, CTO, Digital Asset Holdings. (2016).

[37] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. 2016. Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts. In *IEEE Symposium on Security and Privacy*.

[38] Leslie Lamport. 1998. The part-time parliament. In *TOCS*.

[39] Denis Lukianov. 2015. Compact Confidential Transactions. http://voxelsoft.com/dev/cct.pdf. (2015).

[40] Matteo Maffei, Giulio Malavolta, Manuel Reinert, and Dominique Schröder. 2015. Privacy and access control for outsourced personal records. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 341–358.

[41] Gregory Maxwell. 2013. CoinJoin: Bitcoin privacy for the real world. bitcointalk.org. (August 2013).

[42] Gregory Maxwell. 2013. Confidential Transactions. https://people.xiph.org/~greg/confidential_values.txt. (2013).

[43] Gregory Maxwell and Andrew Poelstra. 2015. Borromean Ring Signatures. https://github.com/Blockstream/borromean_paper. (2015).

[44] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*.

[45] Sarah Meiklejohn, Marjori Pomarole, Grant Jordan, Kirill Levchenko, Damon McCoy, Geoffrey M Voelker, and Stefan Savage. 2013. A fistful of bitcoins: characterizing payments among men with no names. In *IMC*.

[46] Ian Miers, Christina Garman, Matthew Green, and Aviel D Rubin. 2013. Zerocoin: Anonymous Distributed E-Cash from Bitcoin. In *IEEE Symposium on Security and Privacy*.

[47] Andrew Miller, Malte Möser, Kevin Lee, and Arvind Narayanan. 2017. An Empirical Analysis of Linkability in the Monero Blockchain. (2017). http://arxiv.org/abs/1704.04299

[48] Malte Möser. 2013. Anonymity of Bitcoin Transactions: An Analysis of Mixing Services. In *Münster Bitcoin Conference*.

[49] Daniel Mulligan. 1998. Know Your Customer Regulations and the International Banking System: Towards a General Self-Regulatory Regime. *Fordham Int'l LJ* 22 (1998), 2324.

[50] Satoshi Nakamoto. 2009. Bitcoin: A Peer-to-Peer Electronic Cash System. http://bitcoin.org/bitcoin.pdf. (2009).

[51] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *USENIX ATC*.

[52] Torben Pryds Pedersen. 1991. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO*.

[53] Vinay Phegade and Juan Del Cuvillo. 2013. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM Press, New York, New York, USA, 1–1.

[54] Tim Ruffing, Pedro Moreno-Sanchez, and Aniket Kate. 2014. CoinShuffle: Practical Decentralized Coin Mixing for Bitcoin. In *ESORICS*.

[55] Claus-Peter Schnorr. 1991. Efficient signature generation by smart cards. *Journal of cryptology* 4, 3 (1991), 161–174.

[56] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. 2011. Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost. In *ASIACRYPT*.

[57] Jon Southurst. 10 June 2014. Blockchain's SharedCoin Users Can Be Identified, Says Security Expert. *CoinDesk* (10 June 2014).

[58] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM – an Extremely Simple Oblivious RAM Protocol. In *CCS*.

[59] Florian Tramèr, Fan Zhang, Huang Lin, Jean-Pierre Hubaux, Ari Juels, and Elaine Shi. 2017. Sealed-Glass Proofs: Using Transparent Enclaves to Prove and Sell Knowledge. In *IEEE European Symposium on Security and Privacy (Euro S&P)*.

[60] Matthew Trudeau. 2016. Personal communication with Matthew Trudeau, President, TradeWind Markets. (2016).

[61] Luke Valenta and Brendan Rowan. 2015. Blindcoin: Blinded, accountable mixes for Bitcoin. In *Financial Cryptography*. 112–126.

[62] Paul Walker and Phil J. Venables. 19 November 2015. Cryptographic Currency For Securities Settlement. U.S. Patent Application 20150332395. (19 November 2015).

[63] Xiao Shaun Wang, T-H. Hubert Chan, and Elaine Shi. 2015. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *CCS*.

[64] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. 2016. Town Crier: An Authenticated Data Feed for Smart Contracts. In *ACM CCS*.

A CRYPTO PRIMITIVES

We now describe the basic cryptographic primitives used in Solidus. These primitives operate over a multiplictive cyclic group $G = \langle g \rangle$ of order p determined by (linear in) security parameter λ. As we explain, our building blocks require that the Decisional Diffie-Hellman assumption hold for G. (To prevent sub-group attacks using the Pohlig-Hellman algorithm, p is typically prime.) In our implementation of Solidus, G is the secp256k1 elliptic curve group.

A.1 El Gamal Encryption and Account-Balance Representation

The El Gamal cryptosystem (Gen, Enc, Dec) is as follows:

- Gen: $x \xleftarrow{\$} \mathbb{Z}_q$, $\mathsf{sk} \leftarrow x$, $\mathsf{pk} \leftarrow g^x$, output $(\mathsf{pk}, \mathsf{sk})$
- $\mathsf{Enc}(\mathsf{pk}, m)$: if $\neg(m, \mathsf{pk} \in G)$, output $\perp$; $r \xleftarrow{\$} \mathbb{Z}_q$, $\alpha \leftarrow m \cdot \mathsf{pk}^r$, $\beta = g^r$, output $c = (\alpha, \beta)$
- $\mathsf{Dec}(\mathsf{sk}, (\alpha, \beta))$: if $\neg(\mathsf{sk} \in \mathbb{Z}_p \wedge \alpha, \beta \in G)$, output $\perp$; output $\alpha/\beta^{\mathsf{sk}}$

If the Decisional Diffie-Hellman (DDH) problem is hard for G, then El Gamal encryption is semantically secure. El Gamal ciphertexts are malleable, however, a useful feature in our constructions. Specifically, El Gamal has a few useful homomorphisms. Let $(\alpha, \beta) \mapsto m$ mean that (α, β) decrypts to m, i.e., $(\alpha, \beta) = (m \cdot \mathsf{pk}^r, g^r)$ for $r \in \mathbb{Z}_p$. Then the following hold:

- *Multiplicative homomorphism:* $(\alpha, \beta) \mapsto m$, $(\alpha', \beta') \mapsto m'$ implies $(\alpha\alpha', \beta\beta') \mapsto mm'$.
- *Additive homomorphism in exponent space:* $(\alpha, \beta) \mapsto g^m$, $(\alpha', \beta') \mapsto g^{m'}$ implies $(\alpha\alpha', \beta\beta') \mapsto g^{m+m'}$.
- *Multiplicative homomorphism in exponent space:* $(\alpha, \beta) \mapsto g^m$ implies $(\alpha^k, \beta^k) \mapsto g^{mk}$.

Observe that re-encryption of a ciphertext $(\alpha, \beta) \mapsto m$ without knowledge of sk is achievable using the multiplicative homomorphism: Let $r \xleftarrow{\$} \mathbb{Z}_p$, compute a fresh ciphertext $(\alpha', \beta') = (\mathsf{pk}^r, g^r) \mapsto 1$, and then let $(A, B) = (\alpha\alpha', \beta\beta')$. Observe that $(A, B) \mapsto (m \times 1) = m$.

Account-Balance Representation. The cryptographic primitives in Solidus rely on a representation of account balances in the exponent space in order to leverage the additive homomorphism in the exponent space illustrated above. Thus an account balance $\$v$ is encoded as $g^{\$v}$ and represented in an El Gamal ciphertext as $(g^{\$v}\mathsf{pk}^r, g^r)$ for some $r \in \mathbb{Z}_p$. Decrypting a balance thus requires finding the discrete log of $g^{\$v}$. While in general this is hard in G, if $\$v$ is known to be small (e.g., $0 \leq \$v < 2^{30}$), then the balance can be decrypted using a lookup table of manageable size.

A.2 Generalized Schnorr Proofs (GSPs)

Generalized Schnorr Proofs [16] are a type of Σ-protocol, that is, 3-move honest-verifier zero-knowledge (HVZK) proofs (often more specifically defined as special 3-move HVZK proofs with special soundness) [24]. GSP specifically operate over groups for which the discrete log problem and variants are hard. We note that here we consider GSPs only in a cyclic group of prime order, avoiding the caveats of [16] regarding composite-order groups.

Given $x \xleftarrow{\$} \mathbb{Z}_p$ and $y \leftarrow g^x$, there is a simple Σ-protocol to prove knowledge of x to a verifier that knows only $y = g^x$:

- Prover P selects $r \xleftarrow{\$} \mathbb{Z}_p$ and sends $e = g^r$ to Verifier V
- V selects $c \xleftarrow{\$} \mathbb{Z}_p$
- P replies with $s = cx + e$.

Verifier V then checks that $g^s = ey^c$. This protocol is specified in the language of GSPs using notation introduced in [18] as:

$$\mathsf{PoK}\,(x : y = g^x)\,,$$

and is a form of the Schnorr identification protocol.

A more general GSP is possible of the form:

$$\mathsf{PoK}\,(x_1, \ldots, x_k : \mathsf{Pred}(y, (x_1, \ldots, x_k), (y_1, \ldots, y_k)))\,,$$

where Pred is a predicate $y = y_1^{x_1} \cdots y_k^{x_k}$ for a collection of values $y, y_1, \ldots, y_k \in G$ known to the verifier and where the prover aims to prove knowledge of $x_1, \ldots, x_k \in \mathbb{Z}_p$.

It is possible to construct efficient GSPs on conjunctions and disjunctions of such predicates. Additionally, the Fiat-Shamir heuristic [29] can convert GPSs into NIZKs in the Random Oracle Model (ROM) by hashing the prover's message to obtain the challenge. It is also possible to append a supplementary value, which we call a *tag*, to the message to be hashed. The NIZK version of $\mathsf{PoK}(x : y = g^x)$, with tag m, for example, is a Schnorr signature on m. In Solidus, all ZPKs are such NIZKs, a fact we leave implicit in the remainder of the appendix.

A.3 Hidden-Public-Key Signatures

In order to authenticate transactions without revealing the sending user, Solidus employs a *hidden-public-key* (HPK) signature scheme. This simple scheme allows a signer to sign with respect to a signing public key pk that is (El Gamal) encrypted under a bank's public key ePK, i.e., a ciphertext $c \xleftarrow{\$} \mathsf{Enc}(\mathsf{ePK}, \mathsf{pk})$. An HPK signature scheme (hGen, hSign, hVer) with public key ePK is as follows:

- hGen: $\mathsf{sk} \xleftarrow{\$} \mathbb{Z}_q$, $\mathsf{pk} \leftarrow g^{\mathsf{sk}}$, output (pk, sk)
- hSign(sk, ePK, m): $r \xleftarrow{\$} \mathbb{Z}_p$, $(\alpha, \beta) \leftarrow (\mathsf{pk} \cdot \mathsf{ePK}^r, g^r)$. Construct a NIZK
$$pf = \mathsf{PoK}\left((\mathsf{sk}, r) : \left(g^{\mathsf{sk}} \cdot \mathsf{ePK}^r = \alpha\right) \wedge (g^r = \beta)\right)$$
with tag m. Output $\sigma = (c = (\alpha, \beta), pf)$.
- hVer(ePK, m, σ): Parse $\sigma = (c, pf)$ and verify pf with ePK, m, c.

An HPK of this form is not terribly useful in and of itself, as the receiver knows only that a valid signature was generated with respect to *some* key, but learns nothing about the key.

The fact that c is an El Gamal ciphertext of pk under ePK, however, makes such signatures useful in two ways. First, when $\mathcal{U}$ requests a transaction, it allows $\mathcal{B}$ to decrypt pk and identify $\mathcal{U}$. Second, it allows $\mathcal{B}$ to generate a plaintext equivalence proof on c and the encryption of the updated account's key. This second property verifies that the user whose balance is updated knows sk, which thus makes this a valid signature.

A.4 El Gamal Swaps

The vast majority of the computation required for proof generation and verification in Solidus is devoted to what we call *El Gamal swaps*. The operation **ElGamal-Swap** takes as input an ordered pair of El Gamal ciphertexts $(c_0, c_1) = \big((\alpha_0, \beta_0), (\alpha_1, \beta_1)\big)$, a corresponding public key pk, and a value $s \in \{\mathsf{Swap}, \mathsf{NoSwap}\}$. It outputs a fresh ordered pair $\big((\alpha_0', \beta_0'), (\alpha_1', \beta_1')\big)$, re-encrypted under pk, with the same underlying plaintexts. If $s = \mathsf{NoSwap}$, the plaintext order is the same as the original ciphertexts, otherwise it is swapped. The algorithm is as follows:

```
Algorithm ElGamal-Swap($(c_0, c_1)$, pk, $s$):
    parse $(c_0, c_1) = \big((\alpha_0, \beta_0), (\alpha_1, \beta_1)\big)$;
    $r_0 \xleftarrow{\$} \mathbb{Z}_p$, $r_1 \xleftarrow{\$} \mathbb{Z}_p$;
    if $s = \mathsf{NoSwap}$
        $c_0' = (\alpha_0', \beta_0') \leftarrow (\alpha_0 \mathsf{pk}^{r_0}, \beta_0 g^{r_0})$;
        $c_1' = (\alpha_1', \beta_1') \leftarrow (\alpha_1 \mathsf{pk}^{r_1}, \beta_1 g^{r_1})$
    else    // $s = \mathsf{Swap}$
        $c_0' = (\alpha_0', \beta_0') \leftarrow (\alpha_1 \mathsf{pk}^{r_1}, \beta_1 g^{r_1})$;
        $c_1' = (\alpha_1', \beta_1') \leftarrow (\alpha_0 \mathsf{pk}^{r_0}, \beta_0 g^{r_0})$;
    output $(c_0', c_1')$
```

It is possible to prove correct execution of **ElGamal-Swap** for an input / output pair (c_0, c_1) and (c_0', c_1') via a GSP specified in [34].

In Solidus, due to the fact that an account is represented by a pair of ciphertexts on the public key of an account and the account balance, we in fact need perform *double* El Gamal swaps, meaning that two pairs of ciphertexts are swapped using the same value of s. The proof of correctness involves a straightforward extension of the GSP for a single swap.

A double swap proof requires 13 elliptic curve multiplications, while verification requires 18.

A.5 Range Proofs

There are a number of protocols (e.g., [13]) for proving statements of the form $\mathsf{PoK}(x : y = g^x \wedge l_0 \leq x \leq l_p)$.

Drawing on the conceptually simple Confidential Transactions approach [42], we use a GSP to prove that an El Gamal ciphertex $(\alpha, \beta) = (g^{\$v} \mathsf{pk}^r, g^r)$ encrypts an account balance $\$v \geq 0$. To prevent modular wraparound, we specifically prove that $\$v \in [0, 2^t)$ for some small integer t. In our prototype, we set $t = 30$.

The GSP we use to accomplish this operates on each bit of $\$v$ separately. For ciphertext (α_i, β_i), to show that $(\alpha_i, \beta_i) \mapsto \$v_i \in \{g^0, g^{2^i}\}$ under public key pk, it suffices to prove:

$$\mathsf{PoK}\left(r_i : \left((\alpha_i / g^{2^i} = \mathsf{pk}^{r_i}) \vee (\alpha_i = \mathsf{pk}^{r_i})\right) \wedge \beta_i = g^{r_i}\right).$$

Thus the GSP

$$\mathsf{PoK}\left(\{r_i\}_{i=1}^{t} : \bigwedge_{i=0}^{t-1} \left(\left(\alpha_i / g^{2^i} = \mathsf{pk}^{r_i}\right) \vee \left((\alpha_i = \mathsf{pk}^{r_i}) \right) \wedge (\beta_i = g^{r_i}) \right)\right)$$

proves that $(\alpha, \beta) = \left(\prod_{i=1}^{t} \alpha_i, \prod_{i=1}^{t} \beta_i\right)$. Thus if $(\alpha, \beta) \mapsto g^{\$v}$, it must be that $\$v \in [0, 2^t)$ as desired.

This range proof requires $5+10t$ elliptic curve multiplications and t encryptions (requiring 2 multiplications each unless precomutation is employed), while verification requires $7+12t$ multiplications.

We denote such a proof that ciphertext c encrypts a value in $[0, 2^t)$ (in exponential space) by $\mathsf{RangePf}(c, t)$.

B SOLIDUS PVORM CONSTRUCTION

We now present the details of the PVORM construction used in Solidus. The extended paper [21] contains proofs that it is correct, oblivious, and public verifiable. Similar techniques allow construction of a PVORM from any ORAM, ZK proof system, and encryption scheme. Our PVORM is constructed to ensure efficient proof computations in support of high throughputs. For this purpose, we use Circuit ORAM, non-interactive Generalized Schnorr Proofs, and El Gamal encryption.

Circuit ORAM consists of a binary tree of buckets, each containing a fixed number of data blocks. Each location contains an encryption of either a data block or a dummy value. Each logical data block is associated with a single leaf in the tree and physically resides somewhere along the path to that leaf. To access a logical data block, a client reads all blocks along the path to the associated leaf. The client then associates the accessed logical block with a new random leaf, and writes new encryptions of all accessed physical blocks and two other (deterministic) tree paths. During these writes, the client pushes (evicts) existing data blocks as far as possible towards leaves while ensuring that each real data block remains on the path to its associated leaf. These evictions can be done with a number of pairwise swaps of physical memory locations linear in the depth of the tree. We take advantage of the ability to do evictions via pairwise swaps in our PVORM construction.

B.1 Construction

In Solidus, each bank maintains its own PVORM to store user account balances. Since the PVORM is uniquely associated with a single bank, we a simple El Gamal key pair for the key pair specified in Section 4. Each logical address is specified by an account ID and each data block is itself an account balance. To store these, each data block contains a pair of El Gamal encryptions: one of the account ID and one of the balance. We limit the maximum balance to a relatively small value (e.g., 2^{30} or 2^{40}). This allows us to encrypt balances in exponential space, creating an additive homomorphism, while still permitting decryption (using a lookup table). Let t denote the binary log of the maximum balance.

Thus we interpret M as a map from account IDs to account balances. We define the PVORM update function $f((\mathsf{id}, \$v), M)$ that replaces $M[\mathsf{id}]$ with $M[\mathsf{id}] + \$v$ if id $(M[\mathsf{id}] + \$v) \in [0, 2^t)$ and is a key in M. Otherwise f is undefined. Intuitively, f updates a single account balance to any value within the valid range.

As noted in Section 4, we use a fixed-size public stash instead of the dynamic private one assumed by Circuit ORAM. For simplicity, we merge this stash into the root node of the tree. Each data block in the stash is of the same form as those in the tree. We also employ a distinguished fixed block that exists as a single deterministic block on every path. It may be part of the root/stash or separate.

We now specify the implementation of the operations in Section 4. Let (Gen, Enc, Dec) be the standard El Gamal cryptosystem.

Construction 1 (Solidus PVORM)**.** We always initialize all balances to 0. The update space U consists of account ID/transaction value pairs, with values being between the max balance and its negative. Initialization proceeds as follows:

```
Init($1^\lambda$, $\{\mathsf{id}_i\}_{i=1}^{n}$, 0, $U$):
  $(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{Gen}(1^\lambda)$
  for $i \in [1, n]$
    Insert $(\mathsf{id}_i, 0)$ into a Circuit ORAM tree
  Set all unused blocks to $(0, 0)$
  for each block $(\mathsf{id}, 0)$
    Set $C$ at that location to $(\mathsf{Enc}(\mathsf{pk}, \mathsf{id}), \mathsf{Enc}(\mathsf{pk}, 0))$
    Let $(\alpha, \beta)$ be the encryption of 0
    $pf = \mathsf{PoK}(x : (\alpha = \beta^x) \wedge (\mathsf{pk} = g^x))$
  return $(\mathsf{pk}, \mathsf{sk}, C, \{pf\})$
```

If $M = \mathsf{Read}(\mathsf{sk}, C)$, then $\mathsf{Update}(\mathsf{sk}, u, C)$ is only defined when $f(u, M)$ is defined. This property is easy to check given u, sk, and C, so we omit explicit validation. Let B^F be the distinguished fixed block and assume for simplicity that pk is derivable from sk.

```
Update(sk, $u$, $C$):
  $e = (e_{\mathsf{id}}, e_v) \xleftarrow{\$} (\mathsf{Enc}(\mathsf{pk}, \mathsf{id}), \mathsf{Enc}(\mathsf{pk}, \$v))$
  for each block $B_i$ along the path associated with id:
    Let $s$ = Swap if the ID in $B$ is id and NoSwap otherwise.
    $(B^F, B_i') \xleftarrow{\$}$ ElGamal-Swap$((B^F, B_i), \mathsf{pk}, s)$
    $pf_i$ = proof of correct swap
  Let $(c_{\mathsf{id}}, c_v) \leftarrow B^F$
  $rangePf = \mathsf{RangePf}(c_v - e_v, t)$ // (see Appendix A.5)
  Let $(\alpha, \beta) = (c_{\mathsf{id}} - e_{\mathsf{id}})$
    $idPf = \mathsf{PoK}(x : (\alpha = \beta^x) \wedge (\mathsf{pk} = g^x))$
  $B^F \leftarrow (c_{\mathsf{id}}, c_v - e_v)$
  for each block $B_i$ along the eviction paths in Circuit ORAM
    Let $s$ = Swap or NoSwap as per Circuit ORAM
    $(B^F, B_i') \xleftarrow{\$}$ ElGamal-Swap$((B^F, B_i), \mathsf{pk}, s)$
    $pf_i$ = proof of correct swap
  return $(C', e, (\{B_i'\}, \{pf_i\}, rangePf, idPf))$
```

Verification is performed simply by verifying all NIZKs included in the output of Update and by verifying that the updated B^F was computed correctly between the two sets of swaps.

C VARIANTS

We now present three variants on the Solidus system based on different architectural primitives. They provide different guarantees and features which we believe are relevant.

C.1 zk-SNARK PVORM

Though GSPs are highly efficient to construct, they can be quite large and expensive to verify. In circumstances where the size of proofs or the verification time is more important than generation time, zk-SNARKs provide a good alternative. While we could implement the Circuit ORAM-based PVORM described in Section 4 and Appendix B using zk-SNARKs, the large numbers of reencryptions would result in very expensive proofs, even if we were to use symmetric-key primitives. Instead, in Section 7.3 we describe and evaluated a different construction, based on a Merkle tree, which is much more efficient for zk-SNARKs than use of Circuit ORAM.

Our evaluation in Table 1 shows the performance for a single bank update at 128-bit security level, using `libsnark` [10] as the back end for computing the zk-SNARK proofs. The Merkle tree has depth 15 giving the PVORM a capacity of 2^{15} (the same as in our GSP tests). Our implementation includes zk-SNARK-optimized SHA-256 circuits for the Merkle tree, and optimized circuits for RSA-3072 encryption (RSAES-PKCS1-v1_5) and signatures (RSASSA-PKCS1-v1_5 with SHA-256). We used PKCS #1 v1.5 primitives instead of the more up-to-date PKCS #1 v2.2 primitives and alternative public-key schemes for three reasons: they yield less expensive zk-SNARK circuits, they are still used in practice, and they provide a conservative (i.e. competitive) comparison point for GSPs.

When used in Solidus, the zk-SNARK PVORM construction has the clear drawback that the ledger does not contain each user's account balance, even in encrypted form. To compute a user's balance, an auditor would need to parse the transaction ciphertexts, decrypt them and perform all the operations. To reduce such overhead in practice, however, the bank may periodically checkpoint balances. Specifically, it may submit an encrypted version of the Merkle tree leaves, and prove that the encryptions are consistent with a published Merkle tree digest using another zk-SNARK proof. Such a proof is quite expensive to construct, and could only be done periodically, e.g., once per day, without significantly affecting the system throughput. But as transactions are accompanied by ciphertexts, an auditor can start at a checkpoint and then decrypt all subsequent transactions to learn current account balances.

Of course, proof generation times are more important in the applications targeted by Solidus, and in our discussions with blockchain industry technologists, the engineering complexity of zk-SNARKs and trusted setup make them less viable than GSPs today. But zk-SNARKs offer an interesting alternative construction and illustrate what could be a valuable point in the PVORM design space.

C.2 Use of Trusted Hardware

Using Intel Software Guard Extensions (SGX) it is possible to construct a much more efficient PVORM. SGX provides a new set of instructions that permits execution of an application inside an *enclave* [4, 44, 53], which protects the application's control-flow integrity and confidentiality against even a hostile operating system. SGX additionally enables generation of *attestations* that prove to a remote party that an enclave is running a particular application (identified as a hash of its build memory).

To reduce the expense of attestations, an enclave can generate a signing key pair and attest to the integrity of the public key [33, 64]. It can then generate the equivalent of a NIZK by simply signing an assertion that it knows a witness to the statement. Trust in SGX then translates to trust in the application and thus its assertions. Verifying an assertion requires only a single digital signature verification.

Using an SGX-based approach, we can build an extremely fast PVORM. We replace the public-key encryption with symmetric-key encryption and all NIZKs with SGX-signed assertions. We can even employ write-only ORAM to further improve performance. Additionally, a PVORM constructed in the *Sealed-Glass Proof* (SGP) model [59] provides security against arbitrarily strong side-channel attacks, provided that the secret signing key remains protected—such as by using a side-channel-resistant crypto library.

While several complications remain to be address (e.g., the need to share keys across enclaves on different hosts in case of failure), we believe that this approach is eminently practical—albeit under the (strong) assumption of trust in Intel and SGX's implementation.

C.3 Use of Pedersen Commitments

One of the important features of Solidus is auditability, which is greatly aided by having all account balances encrypted on the ledger. Many financial companies and regulatory agencies are, however, wary to include this information, even in encrypted form [12, 36, 60]. While we believe it would degrade the functionality significantly to omit these encryptions, it is not particularly difficult.

Instead of including encrypted balances on the ledger, banks could instead represent PVORM elements as Pedersen commitments [52]. Unlike El Gamal ciphertexts, Pedersen commitments are perfectly hiding and computationally binding. To implement this, banks would need to retain witnesses for each commitment, which consists of both the account balance and the randomization factor. The bank could then reveal this witness to an auditor in order to prove an account balance, and the proof schemes used with El Gamal ciphertexts would require only slight modification to prove information about the known witnesses.