Solidus: Confidential Distributed Ledger Transactions via PVORM

Ethan Cecchetti[⋆,§] ethan@cs.cornell.edu
Fan Zhang[⋆,§] fanz@cs.cornell.edu
Yan Ji[⋆⋆,§] jyamy42@gmail.com
Ahmed Kosba[†,§] akosba@cs.umd.edu
Ari Juels[‡,§] juels@cornell.edu
Elaine Shi[⋆,§] runting@gmail.com

[⋆]Cornell University [⋆⋆]Shanghai Jiao Tong University[*]
[†]University of Maryland [‡]Cornell Tech, Jacobs Institute
[§]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.

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 [22] 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 of services and users [35]. Concerns over this leakage are driving many financial institutions to explore solutions where only digests are stored on-chain and transactions take place elsewhere [1–3]. Such architectures reduce blockchains to little more than a timestamping service; by discarding the vision of a centralized authoritative ledger, they strip blockchains of many of their key benefits. For instance, off-chain transactions are scattered across multiple databases, complicating auditing and data preservation.

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 are appealing to some, but 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 [1, 2]. In such systems a small number of entities—which we call *banks*—manage transactions of some on-chain asset 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 there is little previous investigation of bank-intermediated systems, work on coin mixes and the recently deployed Zcash—based on Zerocash [7]—do improve confidentiality, but with notable limitations. Coin mixes provide only partial confidentiality [35]. Zcash, which relies on zero-knowledge Succinct Non-interactive ARguments of Knowledge (zk-SNARKs) [8] for anonymity, involves reported proof generation times of over a minute on a single consumer machine [7]. While such computation is feasible for a single client performing infrequent transactions, it is prohibitive for a bank in a bank-intermediated system with hundreds of transactions per second. zk-SNARKs also require an un-

[*]Work done at Cornell University.

desirable trusted setup and introduce engineering complexity and cryptographic hardness assumptions that financial institutions are reluctant to embrace [2].

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, Solidus can use efficient NIZK proofs based on Generalized Schnorr Proofs (GSPs) [14, 16]. GSPs are more efficient to construct that 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 and not with 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—something that Zcash, for instance, does not allow. It additionally permits decryption of balance by authorized parties and allows users to prove their own balances if, for example, they wish to transfer funds away from an unresponsive bank.

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 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 full implementation of Solidus and present the results of benchmarking experiments.

Our results are not just a new technical approach to transaction-graph confidentiality on ledgers. They also demonstrate 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 explain some background on the financial system upon which bank-intermediated systems are modeled 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

[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 [35].

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 [35]—to link to real-world identities.

To conceal balances and transaction values, Maxwell proposed a scheme called Confidential Transactions (CT) [31]. 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 a variant of CT that operates on El Gamal ciphertexts to conceal transaction values.

Another approach is Zcash, which provides a privacy overlay on Bitcoin. In brief, it maintains a Merkle tree over commitments to coins. A commitment to a coin owned by $\mathcal{U}$ with a public/private key pair $(\mathsf{pk}, \mathsf{sk})$ includes pk, some randomness r, and a unique serial number. To spend the coin via an operation called *pour*, $\mathcal{U}$ proves in zero-knowledge that the commitment is present in a leaf of the Merkle tree, and that she knows r and sk. Pour additionally reveals the serial number so the coin cannot be double-spent. Zcash proofs use zk-SNARKs, a general-purpose technique for ZK proofs.

Zcash does provide strong confidentiality; balances, transfer amounts, and the transaction graph all remain hidden. Unfortunately, it misaligns with financial settlement systems, as discussed below, and has several serious drawbacks. First, the Merkle tree in Zcash—the authoritative state—grows linearly with total system transaction history. In systems with high transaction throughput this will quickly degrade performance. Second, practical zk-SNARKs require a *trusted setup* which, if subverted, allows proof forgery and theft of coins. Finally, generating proofs in Zcash is extremely expensive—reportedly averaging over one minute on a single consumer machine [7]. This overhead is prohibitive in a bank-intermediated systems such as Solidus. These drawbacks and a desire to align with financial industry needs motivate our very different approach.

[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.

2.2 Bank-intermediated Systems

Managing assets on ledgers is an appealing option for 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 [36], 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 and efficiently store data on untrusted servers. The client maintains a map from logical memory addresses to remote physical addresses and performs reads and writes remotely. Ensuring freshness, integrity,

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

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 read/write 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 [44, 46], 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 [14, 16] 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 [14]. It is possible, additionally, using the Fiat-Shamir heuristic [23], 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.

[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.

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 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 on a ledger [1]. TradeWind is currently building an electronic communication network (ECN) for physical gold bullion to be traded electronically using a bank-intermediated ledger for trade 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, or buy gold on the TradeWind ECN, remove it from the vault, and sell it elsewhere. In the former case, the custodian must create a record of the new assets in the owner's account at her holding bank. In the latter case, the custodian must destroy the corresponding record.

Holdings are represented on the ledger as asset units—fractional ounces of gold—held by individual users. When a user trades gold on the TradeWind ECN, she authorizes her holding bank to transfer some number of gold units to another user. A holding bank may also provide other services based on a user's account balance, such as holding the gold as collateral against a loan. In such cases the bank may freeze some or all of the user's 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: Solidus is the first on-chain transaction settlement system that 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, 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 mainatins 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 $\mathcal{B}_s$ updates its PVORM to reflect the results of the transaction, deducting $\$v$ from $\mathcal{U}_s$'s balance. The receiving bank then performs a similar update on the receiving user's account, increasing her balance by $\$v$.

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. Specifically, verification of a PVORM operation by the send bank checks three things: the transaction was validly signed by one of that bank's users, $\$v$ was deducted from the same user's balance, and that the user's resulting balance is non-negative. The receiving bank performs a PVORM operation to add $\$v$ to the receiving user's account balance, including a proof of a correct balance update. PVORM proofs in Solidus are generated using Generalized Schnorr Proofs.

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 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 their own signature key pair. Each account is uniquely and publicly associated with one bank, so $\text{bank}(\mathcal{U}_j^i) = \mathcal{B}_i$ is well-defined.

[6] It is sometimes desireable for the 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.

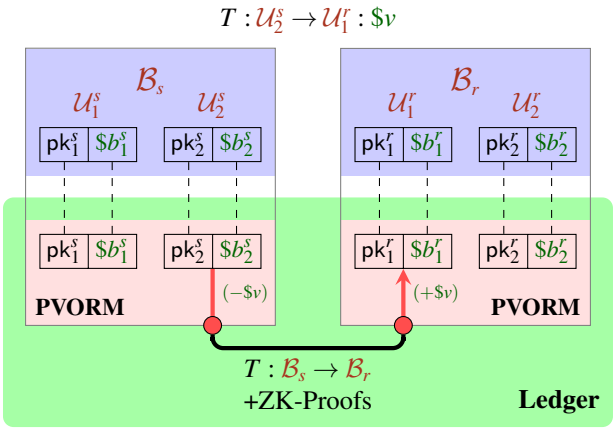

Figure 1: An example transaction T where $\mathcal{U}_2^s$ at $\mathcal{B}_s$ sends $\$v$ to $\mathcal{U}_1^r$ at $\mathcal{B}_r$ and each bank has two users. The upper boxes are the logical (plaintext) memory of each bank's PVORM, and the lower boxes are the associated public (encrypted) memories. Entities 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.

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 $\mathcal{F}_{\text{Ledger}}$, detailed in Appendix C, 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 $\mathcal{U}_{j'}^{i'}$. While this makes sense for ordinary users, financial systems are generally not closed. That is, assets can enter and leave the system through specific channels. To support this, Solidus defines a fixed

set of *asset notaries* $\{\mathcal{U}_1^\$, \ldots, \mathcal{U}_\ell^\$\}$. These are accounts with no recorded balance. If $\mathcal{U}_i^\$$ sends money to $\mathcal{U}_j^i$, there is no need to check $\$b_i^\$$ (which does not exist). To easy auditing of this sensitive action and avoid trying to mimic a regular update for a user with no balance, we simply reveal $\mathcal{U}_i^\$$'s identity.

Asset notaries clearly must be restricted; it would make no sense to allow arbitrary users to create and destroy asset titles. In Example 1, Solidus would designate the custodian as the sole notary that is 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 their own users confidentiality 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.

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 [29], ZooKeeper [25], or Raft [38], a Byzantine consensus protocol such as PBFT [18], a decentralized consensus protocol such as Nakamoto consensus [37], 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 nonfaulty 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 1) a user may wish to use her balance as collateral against a loan. In this case, the bank must be able to ensure that the asset will remain in the user's account until the loan is repaid. The user, meanwhile, may still want a record that she owns the asset [1]. Asset freezing supports this use case.

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 specific audit support, it does enable banks to either prove correct decryption of on-chain data or share their private decryption key with a trusted third party. In the first case, the auditor can, on demand, acquire a transaction log and verify that log's veracity and completeness. In the second case, the auditor can directly and proactively monitor activity within the bank and its accounts.

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). Moreover, any authorization to remove assets from an account can be used only once; there are no replay attacks. Second, no user can spend money she does not have. We implement this guarantee by ensuring that account balances are never negative. Finally, transactions that do not include asset notaries must be zero-sum; the sender's balance always decreases by exactly the same amount that the recipient's balance increases.

To ensure the above properties hold, we require that the correctness of every transaction be proved in a publicly-verifiable fashion (via ZK-Proof, as described above). 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 that they can additionally be verified offline by an auditor.

Confidentiality Guarantees. In order 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, as outlined above. First, account balances are not visible except to the user's bank (and any auditors authorized to read that bank's data). 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. 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. Zcash and similar systems also do not address this concern, but Solidus's superior performance better positions it to support a range of side-channel countermeasures. For example, to eliminate the timing side-channel, Solidus could post transactions at regular intervals in batches of uniform size which are padded out by "dummy" transactions of value 0.

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, C is now publicly visible—such as 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. While traditional ORAMs 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 we cannot use them in the same way. In fact, we base our construction on a general purpose ORAM (Circuit ORAM) as it allows us to implement updates as read-update-write operations. If we used a basic write operation—as would be required with write-only ORAM—we could not prove critical properties about the difference between the old and new 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, as above. 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 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 a set of valid update values U.
- An update constraint function $f(u, M) \to M'$ 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.
- $\mathsf{Ver}(\mathsf{pk}, C, C', e, \mathit{proof}) \to \{\mathsf{true}, \mathsf{false}\}$, a deterministic update verification function.

We also define $\mathsf{Read}(\mathsf{sk}, C) \to M$ and $\mathsf{Dec}(\mathsf{sk}, e) \to u$, 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,

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

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

$$
\begin{array}{l}
\underline{\text{Experiment } \mathbf{Exp}^{\text{Correct}}(\lambda, n, m_0, \{u_i\}_{i=1}^k):} \\
\quad (\mathsf{pk}, \mathsf{sk}, C_0) \xleftarrow{\$} \mathsf{Init}(1^\lambda, n, m_0, U) \\
\quad \textbf{if } \mathsf{Read}(\mathsf{sk}, C_0) \neq M_0, \textbf{return } \mathsf{false} \\
\quad \textbf{for } i = 1 \text{ to } k: \\
\quad\quad (C_i, e_i, proof_i) \xleftarrow{\$} \mathsf{Update}(\mathsf{sk}, u_i, C_{i-1}) \\
\quad\quad \textbf{if } \big[(\mathsf{Read}(\mathsf{sk}, C_i) \neq M_i) \vee (\mathsf{Dec}(\mathsf{sk}, e_i) \neq u_i) \\
\quad\quad\quad \vee \neg \mathsf{Ver}(\mathsf{pk}, C_{i-1}, C_i, e_i, proof_i)\big] \\
\quad\quad\quad \textbf{return } \mathsf{false} \\
\quad \textbf{return } \mathsf{true}
\end{array}
$$

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 gaurantee 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 $negl(\lambda)$ such that for all $n \in \mathbb{N}$, m_0, and U,

$$
\Big| \Pr\Big[\mathbf{Exp}^{\text{Obliv}}(0, \mathcal{A}, \lambda, n, m_0, U) = 1\Big] - \Pr\Big[\mathbf{Exp}^{\text{Obliv}}(1, \mathcal{A}, \lambda, n, m_0, U) = 1\Big] \Big| \leq negl(\lambda)
$$

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

$$
\begin{array}{l}
\underline{\text{Experiment } \mathbf{Exp}^{\text{Obliv}}(b, \mathcal{A}, \lambda, n, m_0, U):} \\
\quad (\mathsf{pk}, \mathsf{sk}, C) \xleftarrow{\$} \mathsf{Init}(1^\lambda, n, m_0, U) \\
\quad \textbf{return } \mathcal{A}^{\mathcal{O}_{b,\mathsf{sk},C}(\cdot,\cdot)}(1^\lambda, \mathsf{pk}, C)
\end{array}
$$

where $\mathcal{O}_{b,\mathsf{sk},C}(\cdot,\cdot)$ is a stateful oracle with initial state $S \leftarrow C$. On input (u_0, u_1), $\mathcal{O}_{b,\mathsf{sk},C}$ executes $(C', e, proof) \xleftarrow{\$} \mathsf{Update}(\mathsf{sk}, u_b, S)$, 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 [43, 44, 46].

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

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

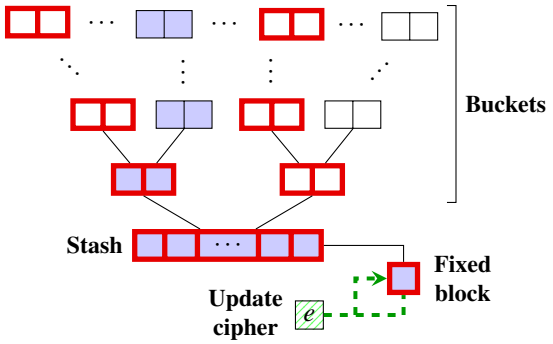

Figure 2: An update for a Circuit ORAM-based PVORM with buckets of size 2. Colors indicate the blocks involved in each operation of the read-update-write structure. Read moves one block from the read path (shaded) into the distinguished fixed block. Then update combines it (homomorphically) with the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

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

$$
\begin{array}{l}
\underline{\text{Experiment } \mathbf{Exp}^{\text{PubVer}}(\mathcal{A}, \lambda, n):} \\
\quad (\mathsf{pk}, \mathsf{sk}, \_) \xleftarrow{\$} \mathsf{Init}(1^\lambda, n, \_, \_); \\
\quad (C, C', e, proof) \xleftarrow{\$} \mathcal{A}(1^\lambda, n, \mathsf{pk}, \mathsf{sk}); \\
\quad \textbf{return } \mathsf{Ver}(\mathsf{pk}, C, C', e, proof) \\
\quad\quad \wedge \big(f(\mathsf{Dec}(\mathsf{sk}, e), \mathsf{Read}(\mathsf{sk}, C)) \neq \mathsf{Read}(\mathsf{sk}, C')\big)
\end{array}
$$

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 [46] with several GSPs. Circuit ORAM places data blocks into buckets organized into a binary tree. It performs updates (evictions) by swapping pairs of blocks along paths in that tree. This structure leads to good performance for two reasons: update operations are logarithmic 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 [27] 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. With the data in a deterministic position, 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 detail this construction and prove that it is correct, oblivious, and publicly verifiable.

Stash Overflow. Circuit ORAM assumes a stash of bounded size, but data loss is possible if the stash overflows. This results 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.

Solidus also employs an bounded stash. The stash must be placed on the ledger, so changing its size would leak information about account locations and thus the transaction graph. In Solidus, however, data loss is catastrophic no matter how infrequent, so we take a different approach to stash overflow. When the stash would overflow, instead of losing data we insert one account deeper into the tree. This insertion is public, so it leaks that regular eviction was insufficient and the location of a single real account (though not the identity of that account).

Solidus inherits the stash overflow probability of Circuit ORAM which is negligible in the stash size [46]. As we show in Section 7, the performance of PVORM updates is linear in the stash size, which gives Solidus a direct performance-privacy trade-off. Moreover, only modest stash sizes make overflow exceedingly unlikely. With buckets of size 3, a stash of size 25 reduces overflow probability to around 2^{-64}.

5 Solidus Protocol

We now present the Solidus protocol. This description relies heavily on cryptographic primitives whose details we defer to Appendix A. We make this choice both to simplify the explanation and to leave operations with several instantiations—such as range proofs—abstract.

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

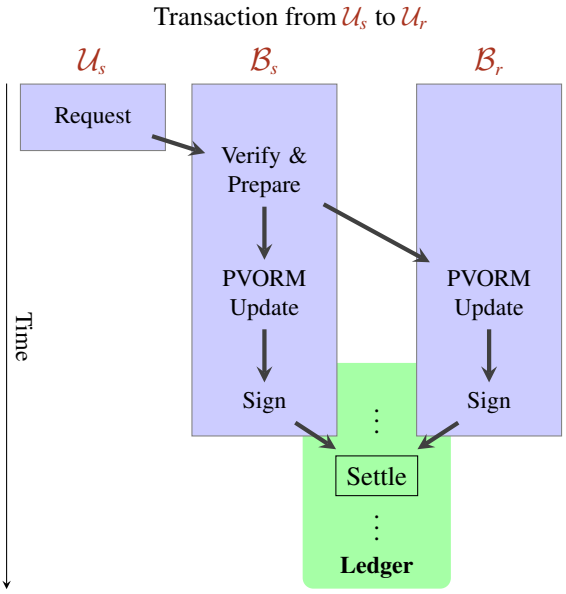

Figure 3: The life cycle of a transaction in Solidus. An arrow from one operation to another means the second depends on the first. Note that $\mathcal{U}_r$ does not appear. The receiving user plays no role in settling transactions.

- 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, signs their updates, and posts 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.

Transaction IDs. To guard against replay attacks, Solidus requires that each transaction have a globally unique transaction ID. If this ID were simply a random bit string, this would require every bank to check each transaction's ID against the IDs of every previous 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 then only valid within a specific window of size T_Δ. For a transaction with $\mathsf{txid} = (T, \mathsf{id})$, if the transaction is processed at time T_{now}, it is only valid if $T_{\text{now}} - T_\Delta < T < T_{\text{now}}$. The timestamp must be in the past, but not by more than T_Δ. This allows a verifying bank to only look back T_Δ when searching for id to guard against 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 Security Definition

We define the security of Solidus in terms of the ideal functionality $\mathcal{F}_{\text{Sol}}$ presented in Figure 4. We define the protocol itself in a hybrid world with an ideal ledger $\mathcal{F}_{\text{Ledger}}$ (see Appendix C). For an adversary $\mathcal{A}$ and environment $\mathcal{Z}$, we let $\mathsf{Hybrid}_{\mathcal{A},\mathcal{Z}}(\lambda)$ denote the hybrid world transcript of $\mathcal{A}$ and $\mathsf{Ideal}_{\mathcal{S},\mathcal{Z}}(\lambda)$ be the transcript produced by a simulator $\mathcal{S}$ run in the ideal world.

Definition 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 $\mathcal{D}$,

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

We assume Solidus employs only universally composable (UC) NIZKs. Prior work [5] 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. This allows us to prove security in the Universal Composability (UC) framework [17].

Theorem 1. *The Solidus protocol satisfies Definition 1 assuming a DDH-hard group in the ROM.*

We provide a proof of Theorem 1 in Appendix C.

$\mathcal{F}_{\text{Sol}}\left[\{\mathcal{B}_i\}_{i=1}^{k}, \{\mathcal{U}_i\}_{i=1}^{n}, \{\mathcal{U}_i^{\$}\}_{i=1}^{\ell}\right]$

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 4: 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.

6 Optimizations

We now present a few optimizations to make Solidus more practical. Some of these optimizations are only appropriate for certain use cases, but they may result 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, the homomorphic properties of El Gamal allow us to re-randomize ciphertexts by combining them with fresh encryptions of the group identity. That is, in a group $G = \langle g \rangle$ of size p, given a public/private key pair $(\mathsf{pk} = g^{\mathsf{sk}}, \mathsf{sk})$ and a ciphertext $c = (\alpha, \beta)$, we can generate a re-randomized c' by picking a random $r \leftarrow \mathbb{Z}_p$ and letting $c' = (\alpha \cdot \mathsf{pk}^r, \beta \cdot g^r)$.

Conveniently, computing (pk^r, g^r) only requires knowledge of the group G, the generator g, and the public key pk, all of which are static for a given bank across the lifetime of the system. This means we can precompute these unit ciphertexts and re-randomize by multiplying in a precomputed value.

As the system admits an unbounded number of transactions over its lifetime, we must continuously generate these randomization factors. Many financial systems have very predictable high and low load times (e.g., traffic is often very light at night), so they can utilize

otherwise-idle hardware to compute randomization factors during low traffic times. If the precomputation process can generate 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 trusted to function as covert, rather than malicious, 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, for example, each Solidus bank is controlled by a large commercial bank. While the bank may wish to learn as much information as possible—including by deviating from the protocol—the cost of being caught misbehaving is high enough to deter attributable protocol deviations.

Under these assumptions, we can omit online verification entirely. If any 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

In a setting where failures of any form are rare (even if Byzantine failures are possible), we can optimize further. Again, this setting is very reasonable in the case of large commercial banks.

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, it can start processing T_2 early while generating proofs for T_1. When faults are rare, this is a reasonable assumption. 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 making this optimization appropriate only when failures are rare.

7 Experiments

We now present performance results for our PVORM and Solidus implementations. For all experiments we ran on `c4.8xlarge` Amazon EC2 instances and employed the precomputation optimization discussed in Section 6.1. These benchmarks do not include time to compute new encryption randomization factors.

7.1 PVORM Performance

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

Bucket and Stash Size. Figure 5 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 the bucket and stash sizes determine the likelihood of stash overflow, this provides a performance-privacy tradeoff.

Tree Depth. Figure 6 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 lends itself to highly parallel operation. A single update contains a large number of NIZKs which can be created or verified independently. Figure 7 shows the performance for a single PVORM with a varying number 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

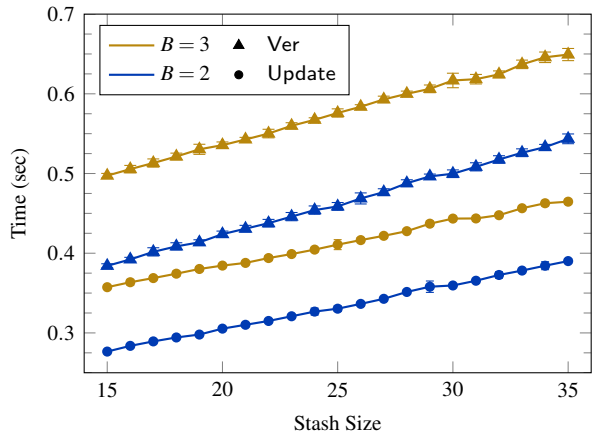

Figure 5: PVORM performance with capacity 2^{15} for buckets of size $B = 2$ and $B = 3$ as stash size varies.

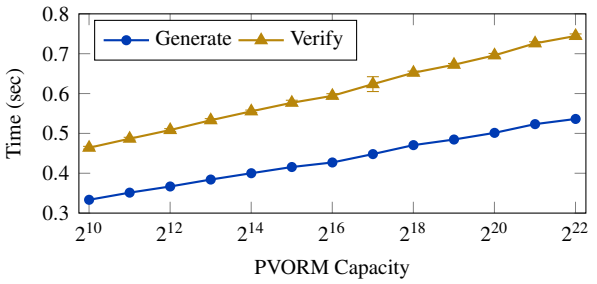

Figure 6: PVORM capacity scaling with buckets of size 3 and stash of size 25.

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 196 KB (or 117 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]Normally 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. Decompression, however, requires nontrivial overhead.

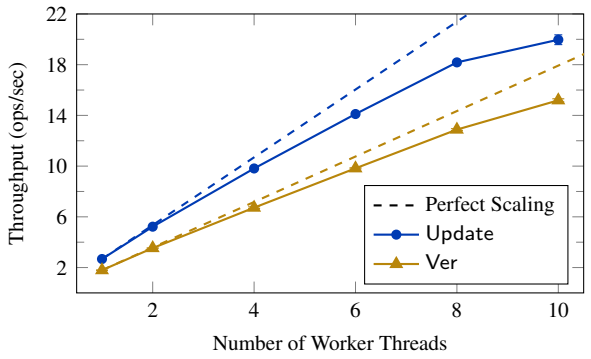

Figure 7: Parallel PVORM performance using size 3 buckets, a size 25 stash, and capacity of 2^{15}. Dashed lines show perfect scaling where all computation is parallelized with no overhead.

7.2 Solidus System Performance

We now present the performance of fully distributed Solidus system tests 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 for reasonable testing and give a stash overflow probability of around 2^{-64}. We maintain the ledger by having each bank's host also run a ZooKeeper [25] node. We make no attempt to tune ZooKeeper or optimize off-ledger communication.

We test this configuration with each bank fully loaded with both incoming and outgoing transactions. As explained in Section 6.2, it is often reasonable to perform some or all transaction verification offline, so we also test performance with online verification turned off.

Figure 8 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 transaction verification cross multiple machines, improving capacity and increasing throughput. Pipelinig 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.) We did not benchmark these options as our prototype does not support distributing a single bank to multiple hosts.

7.3 zk-SNARK Comparison

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

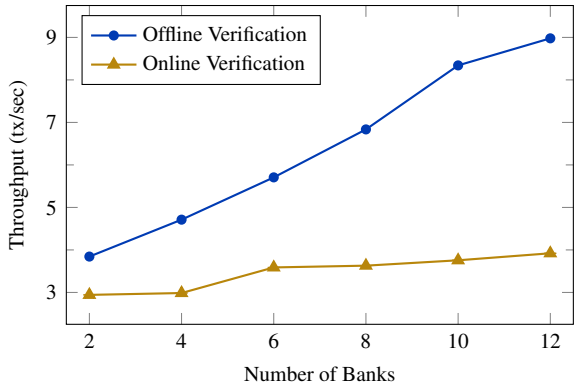

Figure 8: Solidus performance distributed using ZooKeeper. Each bank is a ZooKeeper node and maintains a PVORM with size 3 buckets, a size 25 stash, and capacity 2^{15}.

	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.

Simply taking our Circuit ORAM PVORM construction and converting all proofs to zk-SNARKs would require needless and expensive cryptographic operations. As zk-SNARKs can prove correct application of an arbitrary circuit [8], we use a compact Merkle tree structure which we describe in detail in Appendix D.1.

We implemented this construction using an equivalent security level to that of our GSP-based PVORM.[8] Table 1 shows its performance running on a `c4.8xlarge` EC2 instance. We see that, 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 our work with several other anonymous digital currencies. As we discussed ORAM in Sections 2.3 and 4, we do not do so again here.

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

Anonymous e-cash. Anonymous e-cash was originally proposed by Chaum [19, 20] and later improved by others [13, 15, 24]. Known e-cash schemes rely on centralized trust, meaning that a single authority must be involved in every transaction. In some schemes, a trustee can revoke anonymity from a user or transaction using a private key; this feature permits tracing of illicit activity. Our design goal in Solidus is different. Rather than having to trust a central bank, Solidus users can choose which bank they trust, and anonymity is preserved with respect to all other entities in the system. Solidus users trust their respective banks only for privacy and availability. While a malicious bank can prevent a user from transacting, the requirement for digital signatures from users prevents a bank from abusing users' resources. Additionally, as noted above, a user can prove asset possession with the help of an auditor and move her account to another bank should her funds be inappropriately frozen.

Anonymous Decentralized Cryptocurrencies. Zcash and its antecedent Zerocash [7] provide an anonymous decentralized cryptocurrency. Specifically, it relies on preprocessing zk-SNARKs to ensure conservation of money, prevent double spending, and hide both the transaction value and transaction graph. In Hawk, Kosba et al. point out flaws in the security definitions and proofs in Zerocash and construct their own, similar decentralized anonymous cryptocurrency (with privacy-preserving smart contracts) [28].

Both Zcash and Hawk rely on preprocessing zk-SNARKs, and therefore the system requires trusted setup. While Ben-Sasson et al. point out that trusted setup can be decentralized through multiparty computation [9], this is a complicated process not yet performed in any significant fielded system. 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 aim to provide auditability as Solidus does; as designed, they do not record assets on-chain and only record commitments.

Besides Zcash and Hawk, other schemes [6, 32, 41] provide various forms of mixing for decentralized cryptocurrencies (cryptographic or non-cryptographic) to enhance the anonymity and help obscure the transaction graph. These systems do not provide full security, however, and it may still be possible to break anonymity through statistical analysis. It is not well-understood how much mixing is needed to resist statistical analysis and achieve the levels of anonymity desired in practice. In comparison, Solidus achieves much stronger anonymity guarantees; the source and destination banks are revealed, but no other transaction information is leaked.

Confidential Transactions. A class of schemes often called Confidential Transactions [30, 31, 33] hide transaction amounts, but do not aim to provide transaction graph privacy. (A subset of these schemes using unorthodox cryptographic tools without accompanying proofs has been shown to be flawed and has been going through a "build, break, fix" cycle—e.g., incarnations of [30] were broken twice.) Solidus employs a Confidential Transaction scheme to hide transaction amounts; this scheme is similar to [31], but makes more direct use of and inherits the provable security properties of GSPs.

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, and Isaac Sheff for their insightful comments and help editing.

References

[1] Personal communication with Matthew Trudeau, President, TradeWind Markets.

[2] Personal communication with Shaul Kfir, CTO, Digital Asset Holdings.

[3] Personal communication with Tamás Blummer, Chief Ledger Architect, Digital Asset Holdings.

[4] I. Anati, S. Gueron, and S. Johnson. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[5] B. Barak, R. Canetti, J. B. Nielsen, and R. Pass. Universally Composable Protocols with Relaxed Set-up Assumptions. In *FOCS*, 2004.

[6] S. Barber, X. Boyen, E. Shi, , and E. Uzun. Bitter to Better—How to Make Bitcoin a Better Currency. In *Financial Cryptography*, 2012.

[7] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *IEEE Symposium on Security and Privacy*, 2014.

[8] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. Snarks for C: verifying program executions succinctly and in zero knowledge. In *CRYPTO*, 2013.

[9] E. Ben-Sasson, A. Chiesa, M. Green, E. Tromer, and M. Virza. Secure Sampling of Public Parameters for Succinct Zero Knowledge Proofs. In *IEEE Symposium on Security and Privacy*, 2015.

[10] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture. In *USENIX Security*, 2014.

[11] E.-O. Blass, T. Mayberry, G. Noubir, and K. Onarlioglu. Toward Robust Hidden Volumes Using Write-Only Oblivious RAM. In *CCS*, 2014.

[12] F. Boudot. Efficient proofs that a committed number lies in an interval. In *EUROCRYPT*, 2000.

[13] J. Camenisch, S. Hohenberger, and A. Lysyanskaya. Compact e-cash. In *EUROCRYPT*, 2005.

[14] J. Camenisch, A. Kiayias, and M. Yung. On the Portability of Generalized Schnorr Proofs. In *EUROCRYPT*, 2009.

[15] J. Camenisch, A. Lysyanskaya, and M. Meyerovich. Endorsed E-Cash. In *IEEE Symposium on Security and Privacy*, 2007.

[16] J. Camenisch and M. Stadler. Efficient group signature schemes for large groups. In *CRYPTO*. 1997.

[17] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, 2001.

[18] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *OSDI*, 1999.

[19] D. Chaum. Blind signatures for untraceable payments. In *CRYPTO*, 1982.

[20] D. Chaum, A. Fiat, and M. Naor. Untraceable electronic cash. In *CRYPTO*, 1990.

[21] I. Damgård. On σ-protocols. *Lecture Notes, University of Aarhus, Department for Computer Science*, 2002.

[22] M. del Castillo. Overstock just closed its first day of blockchain stock trading. *Coindesk*, 16 December 2016.

[23] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *EUROCRYPT*, 1986.

[24] G. Hinterwälder, C. T. Zenger, F. Baldimtsi, A. Lysyanskaya, C. Paar, and W. P. Burleson. Efficient E-Cash in Practice: NFC-Based Payments for Public Transportation Systems. In *PETS*, 2013.

[25] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX ATC*, 2010.

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

[27] M. Jakobsson and A. Juels. Millimix: Mixing in small batches. Technical report, DIMACS Technical report 99-33, 1999.

[28] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts. In *IEEE Symposium on Security and Privacy*, 2016.

[29] L. Lamport. The part-time parliament. In *TOCS*, 1998.

[30] D. Lukianov. Compact confidential transactions. `http://voxelsoft.com/dev/cct.pdf`.

[31] G. Maxwell. Confidential transactions. `https://people.xiph.org/~greg/confidential_values.txt`.

[32] G. Maxwell. CoinJoin: Bitcoin privacy for the real world. `bitcointalk.org`, August 2013.

[33] G. Maxwell and A. Poelstra. Borromean ring signatures. `https://github.com/Blockstream/borromean_paper`.

[34] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[35] S. Meiklejohn, M. Pomarole, G. Jordan, K. Levchenko, D. McCoy, G. M. Voelker, and S. Savage. A fistful of bitcoins: characterizing payments among men with no names. In *IMC*, 2013.

[36] D. Mulligan. Know your customer regulations and the international banking system: Towards a general self-regulatory regime. *Fordham Int'l LJ*, 22:2324, 1998.

[37] S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. `http://bitcoin.org/bitcoin.pdf`, 2009.

[38] D. Ongaro and J. Ousterhout. In Search of an Understandable Consensus Algorithm. In *USENIX ATC*, 2014.

[39] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO*, 1991.

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

[41] T. Ruffing, P. Moreno-Sanchez, and A. Kate. CoinShuffle: Practical decentralized coin mixing for Bitcoin. In *ESORICS*, 2014.

[42] C.-P. Schnorr. Efficient signature generation by smart cards. *Journal of cryptology*, 4(3):161–174, 1991.

[43] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost. In *ASIACRYPT*, 2011.

[44] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM – an extremely simple oblivious ram protocol. In *CCS*, 2013.

[45] F. Tramèr, F. Zhang, H. Lin, J.-P. Hubaux, A. Juels, and E. Shi. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. In *IEEE European Symposium on Security and Privacy (Euro S&P)*, 2017. To appear.

[46] X. S. Wang, T.-H. H. Chan, and E. Shi. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *CCS*, 2015.

[47] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi. Town crier: An authenticated data feed for smart contracts. In *ACM CCS*, 2016.

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 $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ 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 in the form $g^{\$v}$ and represented in an El Gamal ciphertext as $(g^{\$v}\mathsf{pk}^r, g^r)$ for some $r \in \mathbb{Z}_p$. Decrypting an account balance thus requires solving the discrete log problem on $g^{\$v}$. While in general this is hard in G, if $\$v$ is known to be relatively 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 [14] 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) [21]. 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 [14] 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 [16] as:

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

and is a form of the Schnorr identification protocol.

A more general GSP is possible of the form:

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

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 that combine such predicates conjunctively and disjunctively, and efficient constructions for other predicates have been shown as well. Additionally, GSPs may be converted in the Random Oracle Model (ROM) into NIZKs using the Fiat-Shamir heuristic [23], which involves hashing the prover's message in the first move of the Σ-protocol. 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, and we leave this fact 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 $(\alpha,\beta) \xleftarrow{\$} \mathsf{Enc}(\mathsf{ePK},\mathsf{pk})$. An HPK signature scheme $(\mathsf{hGen},\mathsf{hSign},\mathsf{hVer})$ with public key ePK is as follows:

- hGen: $\mathsf{sk} \xleftarrow{\$} \mathbb{Z}_q$, $\mathsf{pk} \leftarrow g^{\mathsf{sk}}$, output $(\mathsf{pk},\mathsf{sk})$
- $\mathsf{hSign}(\mathsf{sk},\mathsf{ePK},m)$: $r \xleftarrow{\$} \mathbb{Z}_p$, $(\alpha,\beta) \leftarrow (\mathsf{pk}\cdot\mathsf{ePK}^r, g^r)$. Construct a NIZK

  $$pf = \mathsf{PoK}\left((\mathsf{sk},r) : \left(g^{\mathsf{sk}}\cdot\mathsf{ePK}^r = \alpha\right) \wedge \left(g^r = \beta\right)\right)$$

  with tag m. Output $\sigma = ((\alpha,\beta),pf)$.
- $\mathsf{hVer}(\mathsf{ePK},m,\sigma)$: Parse $\sigma = ((\alpha,\beta),pf)$ and verify pf with ePK, m, (α,β).

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 (α,β) 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 (α,β) and the encrypted account key associated with the balance $\mathcal{B}$ is updated in its PVORM. 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**$((c_0,c_1),\mathsf{pk},s)$:
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 // $s = \mathsf{Swap}$
  $c_0' = (\alpha_0',\beta_0') \leftarrow (\alpha_1 \mathsf{pk}^{r_1}, \beta_1 g^{r_1})$;
  $c_1' = (\alpha_1',\beta_1') \leftarrow (\alpha_0 \mathsf{pk}^{r_0}, \beta_1 g^{r_0})$;
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 [27].

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., [12]) for proving statements of the form $\mathsf{PoK}(x : y = g^x \wedge l_0 \le x \le l_p)$.

In Solidus, drawing on the conceptually simple Confidential Transactions approach [31], we use a GSP to prove of an El Gamal ciphertext $c = (\alpha,\beta) = (g^{\$v}\mathsf{pk}^r, g^r)$ that represents an account balance $0 \le \$v$. Specifically, to preclude modular wraparound, we prove that the integer $\$v \in [0,2^t)$ for parameter t, which determines the upper bound on account balances. In our prototype, we set $t = 30$.

We now describe the GSP we use accomplishes this range proof in a bitwise manner. First, we observe that to show for ciphertext (α_i,β_i) 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

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

proves for $(\alpha,\beta) = \left(\prod_{i=1}^{t}\alpha_i, \prod_{i=1}^{t}\beta_i\right)$ that $(\alpha,\beta) \mapsto g^{\$v}$ such that $\$v \in [0,2^t)$, i.e., that (α,β) is a ciphertext on account balance $\$v \in [0,2^t)$.

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 $\mathsf{RangePf}(c,t)$.

A.6 Circuit ORAM

Solidus's primary data structure used to store account balances on the ledger is a PVORM based on the structure of Circuit ORAM [46]. PVORM, however, aims to provide very different guarantees than classical ORAM. An ORAM enables a *client* with limited local memory to maintain a piece of large virtual memory M in a data structure C outsourced to a more powerful external device generically called a *server*. The goal is to enable the client to store M confidentially *with as little local storage as possible*.

An ORAM ensures *access-pattern confidentiality*; despite its ability to observe the client's accesses to C, the server learns nothing (no non-negligible) information about the client's pattern of access to blocks in M. Blocks in C are encrypted using a *symmetric-key* cipher to ensure data confidentiality. But note that encryption alone does not conceal access patterns. M is structured as a set of *blocks* $M[1],M[2],\ldots,M[N]$. Were $C[\mathtt{idx}]$ simply an encryption of the current value of $M[\mathtt{idx}]$, for instance, then the server would know every time the client reads from or writes to $M[\mathtt{idx}]$, as it would see the client access $C[\mathtt{idx}]$.

Thus, to achieve access-pattern confidentiality, ORAM implementations require a more sophisticated approach.

In this approach, C is represented as a tree of depth $L = \log N + 1$ (N is assumed to be a power of 2). Each node in the tree contains a *bucket* that has B slots for storage of blocks, where B is a system parameter. Most of these slots are empty at a given time, an important fact, as we shall see below.

A block takes the form $\mathtt{idx}\|\mathtt{label}\|\mathtt{data}$, where $\mathtt{idx}$ is the index of a block—the value $\mathtt{idx}$ corresponding to its virtual memory slot $M[\mathtt{idx}]$, $\mathtt{label}$ identifies a leaf in

the tree along the path to which from the root the block is located in C, and `data` stores the block contents.

The client maintains a small amount of local memory called a *stash*, which is a buffer to handle overflow from C. The client also stores a *position map* PosMap, a data structure such that $\mathsf{PosMap}[\mathtt{idx}] = \mathtt{label}$. That is, PosMap maps a given block's index `idx` in M to its corresponding leaf value `label`. (PosMap can be stored recursively in a separate ORAM on the server to reduce storage overhead, a feature that is not relevant to PVORM.)

Reads and writes involve the same basic operation **Access** by the client on C, which is as follows.

Algorithm **Access**(op):

// Note: op = ("read", idx) or ("write", idx, data*)
label ← PosMap[idx];
{idx||label||data} ← ReadAndRm(idx, label);
PosMap[idx] $\xleftarrow{\$}$ $[0, N-1]$;
if op = "read" then data* ← data;
stash.add({idx||PosMap[idx]||data*});
Evict();
output data

Here, ReadAndRm reads the full path in C containing the target block and removes the block (re-encrypting blocks along the path), while stash.add performs the obvious operation of adding a block to the stash. Evict can be implemented either randomly or deterministically. The random approach picks two leaves $leaf_l$ and $leaf_r$ uniformly at random from the left and right halves of the three, respectively, and performs what is called an eviction pass in the root-to-leaf paths they define. The deterministic approach (which we adopt in our PVORM construction) does the same, but it selects $leaf_l$ and $leaf_r$ in a rotating deterministic order designed to place eviction passes on consecutive accesses as far away from each other as possible while still rotating through every leaf over enough accesses.

An eviction pass on a given path involves performing swaps on pairs of adjacent path elements one by one from the top to bottom of the tree, with the stash treated as a special "level 0," i.e., sitting above the root. These swaps aim to move blocks down the path to the lowest possible levels. A block is "picked up" and moved through successive swaps to the lowest point such that it remains on the path defined by `label` and there is an empty slot available for it. At this point it is "dropped"—inserted into the bucket at that level. A block may be picked up from the slot into which the last one was dropped or swapping may continue until another block is reached that can be pushed further down the path. The reason for performing evictions on two paths rather than one is to ensure that blocks remain deep enough globally in C to prevent substantial overflow into the stash.

This processing step in Circuit ORAM is in fact quite complicated. The client does not have full local information about where blocks reside in C, and therefore must plan swaps using metadata. (This complication does not arise in PVORM, however, as we explain below.)

Other tree-based ORAMs, such as Path ORAM [44], differ primarily in their use of alternative eviction strategies. The use of swaps in Circuit ORAM is especially conducive to efficient NIZK production in Solidus, however, which is the reason it is used in the Solidus PVORM.

B Solidus PVORM Construction

We now present the details of the PVORM construction used in Solidus and prove that it is a correct, oblivious, and publicly verifiable PVORM. We note that it is possible to construct a PVORM from any ORAM, ZK proof system, and encryption scheme (symmetric or public-key). Our PVORM in Solidus, however, is constructed to ensure highly efficient proof computations in support of high throughputs. For this purpose, we use Circuit ORAM, non-interactive Generalized Schnorr Proofs, and El Gamal encryption.

Recall from above that 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. In order to access a logical data block (read or write), the 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 out new encryptions of all blocks along the accessed path and two other deterministic paths in the tree. During these writes, the client evicts existing data blocks towards leaves as possible while maintaining the invariant that each real data blocks 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 exists as a key in M and $(M[\mathsf{id}] + \$v) \in [0, 2^t)$. Otherwise $f((\mathsf{id}, \$v), M)$ 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 additionally employ a single distinguished fixed block. This block is simply a single deterministic block that exists on every path. It may be part of the root bucket/stash or it may be its own separate location.

We now describe the implementation of each operation defined in Section 4. Let $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ 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:

$\mathsf{Init}(1^\lambda, \{\mathsf{id}_i\}_{i=1}^n, 0, U)$:
- $(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{Gen}(1^\lambda)$
- **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 (α, β) be the encryption of 0
  - $pf = \mathsf{PoK}(x : (\alpha = \beta^x) \wedge (\mathsf{pk} = g^x))$
- **return** $(\mathsf{pk}, \mathsf{sk}, C, \{pf\})$

Let $M = \mathsf{Read}(\mathsf{sk}, C)$. We note that $\mathsf{Update}(\mathsf{sk}, u, C)$ is only defined when $f(u, M)$ is defined. Given u, sk, and C, this property is easy to check, so our definition simply assumes Update is defined on the inputs and does not check explicitly. Let B^F be the distinguished fixed block. For simplicity assume that the pk associated with sk is available (either by being stored as part of sk or derivable from sk).

$\mathsf{Update}(\mathsf{sk}, u, C)$:
- $e = (e_{\mathsf{id}}, e_v) \xleftarrow{\$} (\mathsf{Enc}(\mathsf{pk}, \mathsf{id}), \mathsf{Enc}(\mathsf{pk}, \$v))$
- **for** each block B_i along the path associated with id:
  - Let $s = \mathsf{Swap}$ 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 $s = \mathsf{Swap}$ 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.

B.2 Security Proofs

We now prove the security of the construction given in the previous section.

Theorem 2 (PVORM Correctness)**.** *Construction 1 is a* correct *PVORM.*

Proof. The following properties ensure correctness.

- Circuit ORAM is correct when the stash does not overflow and Construction 1 modifies Circuit ORAM to leak transaction graph information instead of lose data on overflows.
- El Gamal is correct and includes a multiplicative homomorphism, while we encrypt account balances in exponential space, thus making the homomorphism additive.
- Construction 1 employs correct NIZKs and only attempts to prove true statements.

□

To prove obliviousness, we provide a hardness reduction to the Decisional Diffie-Hellman (DDH) problem. We do this through a series of reductions. First we consider the following classic definition of CPA security that a cryptosystem $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is *CPA secure* if for all

PPT adversaries $\mathcal{A}$ there is a negligible function *negl* such that

$$\left|\Pr\left[\mathbf{Exp}^{\text{CPA}}(0,\mathcal{A},\lambda)=1\right] - \Pr\left[\mathbf{Exp}^{\text{CPA}}(1,\mathcal{A},\lambda)=1\right]\right| \leq negl(\lambda).$$

where $\mathbf{Exp}^{\text{CPA}}(b,\mathcal{A},\lambda)$ is defined as

Experiment $\mathbf{Exp}^{\text{EG-CPA}}(b,\mathcal{A},\lambda)$:
$(\mathsf{sk},\mathsf{pk}) \xleftarrow{\$} \mathsf{Gen}(1^\lambda)$
$(m_0,m_1) \xleftarrow{\$} \mathcal{A}(1^\lambda,\mathsf{pk})$
$c \xleftarrow{\$} \mathsf{Enc}(\mathsf{pk},m_b)$
return $\mathcal{A}(1^\lambda,c)$

It is well known that El Gamal (which Solidus uses) is CPA-secure in a DDH-hard group. We further define *double-CPA security* which we will use to prove obliviousness of our PVORM construction.

Definition 2 (Double-CPA Security)**.** A cryptosystem $(\mathsf{Gen},\mathsf{Enc},\mathsf{Dec})$ is *double-CPA secure* if for all PPT adversaries $\mathcal{A}$ there is a negligible *negl* such that

$$\left|\Pr\left[\mathbf{Exp}^{\text{2CPA}}(0,\mathcal{A},\lambda)=1\right] - \Pr\left[\mathbf{Exp}^{\text{2CPA}}(1,\mathcal{A},\lambda)=1\right]\right| \geq \varepsilon(\lambda).$$

where $\mathbf{Exp}^{\text{2CPA}}(0,\mathcal{A},\lambda)$ is defined as

Experiment $\mathbf{Exp}^{\text{2CPA}}(b,\mathcal{A},\lambda)$:
$(\mathsf{sk},\mathsf{pk}) \xleftarrow{\$} \mathsf{Gen}(1^\lambda)$
$((m_0,m_0'),(m_1,m_1')) \xleftarrow{\$} \mathcal{A}(1^\lambda,\mathsf{pk})$
$c \xleftarrow{\$} \mathsf{Enc}(\mathsf{pk},m_b)$
$c' \xleftarrow{\$} \mathsf{Enc}(\mathsf{pk},m_b')$
return $\mathcal{A}(1^\lambda,c,c')$

We now prove by a hybrid argument that any public-key cryptosystem that is CPA secure (e.g. El Gamal) is double-CPA secure.

Lemma 1 (Double-CPA Security)**.** *Let* $(\mathsf{Gen},\mathsf{Enc},\mathsf{Dec})$ *be a CPA-secure public-key cryptosystem. For all stateful PPT adversaries* $\mathcal{A}$*, there is a negligible function* $negl(\lambda)$ *such that*

$$\left|\Pr\left[\mathbf{Exp}^{2CPA}(0,\mathcal{A},\lambda)=1\right] - \Pr\left[\mathbf{Exp}^{2CPA}(1,\mathcal{A},\lambda)=1\right]\right| \leq negl(\lambda).$$

Proof. Assume for contradiction that there is some $\mathcal{A}$ and non-negligible $\varepsilon(\lambda)$ such that

$$\left|\Pr\left[\mathbf{Exp}^{\text{2CPA}}(0,\mathcal{A},\lambda)=1\right] - \Pr\left[\mathbf{Exp}^{\text{2CPA}}(1,\mathcal{A},\lambda)=1\right]\right| \geq \varepsilon(\lambda).$$

We now consider a set of hybrid experiments. Let $H_0 = \mathbf{Exp}^{\text{2CPA}}(0,\mathcal{A},\lambda)$, $H_2 = \mathbf{Exp}^{\text{2CPA}}(1,\mathcal{A},\lambda)$, and

Experiment H_1:
$(\mathsf{sk},\mathsf{pk}) \xleftarrow{\$} \mathsf{Gen}(1^\lambda)$
$((m_0,m_0'),(m_1,m_1')) \xleftarrow{\$} \mathcal{A}(1^\lambda,\mathsf{pk})$
$c \xleftarrow{\$} \mathsf{Enc}(\mathsf{pk},m_0)$
$c' \xleftarrow{\$} \mathsf{Enc}(\mathsf{pk},m_1')$
return $\mathcal{A}(1^\lambda,c,c')$

Note that we encrypt m_0 (as in H_0) and m_1' (as in H_2). By the standard hybrid argument $\mathcal{A}$ must have advantage at least $\varepsilon(\lambda)/2$ in distinguishing either between H_0 and H_1 or between H_1 and H_2.

We now construct an adversary $\mathcal{B}$ to break the CPA security of $(\mathsf{Gen},\mathsf{Enc},\mathsf{Dec})$. On input $(1^\lambda,\mathsf{pk})$, $\mathcal{B}$ first runs $\mathcal{A}$ to get $(m_0,m_0'),(m_1,m_1')$. It then picks a random $i \xleftarrow{\$} \{0,1\}$. We handle these cases separately.

- $i=0$: In this case $\mathcal{B}$ outputs (m_0,m_1). On receipt of challenge c it computes $c' \xleftarrow{\$} \mathsf{Enc}(\mathsf{pk},m_1')$, submits $(1^\lambda,c,c')$ to $\mathcal{A}$ and returns the result.
- $i=1$: In this case $\mathcal{B}$ outputs (m_0',m_1'). On receipt of challenge c', it computes $c \xleftarrow{\$} \mathsf{Enc}(\mathsf{pk},m_0)$ and submits $(1^\lambda,c,c')$ to $\mathcal{A}$ and returns the result.

In the first case, if c encrypts m_0 then this is exactly experiment H_1 and if c encrypts m_1, this is experiment H_2. For the second case, $\mathcal{B}$ has similarly generated either experiment H_0 or H_1. $\mathcal{B}$ will succeed exactly when $\mathcal{A}$ succeeds. Since $\mathcal{A}$ has advantage at least $\varepsilon(\lambda)/2$ in one of these experiments and $\mathcal{B}$ randomly selects which experiment to run, it must be the case that $\mathcal{B}$ succeeds with advantage at least $\varepsilon(\lambda)/4$, which is non-negligible. By assumption, however, $(\mathsf{Gen},\mathsf{Enc},\mathsf{Dec})$ is CPA-secure, so this contradicts our assumption that $\mathcal{A}$ exists. Thus $(\mathsf{Gen},\mathsf{Enc},\mathsf{Dec})$ is double-CPA secure. □

Theorem 3 (PVORM Obliviousness)**.** *Construction 1 is* oblivious *in the ROM assuming a DDH-hard group.*

Proof. Assume for contradiction that there exists some PPT adversary $\mathcal{A}$ and non-negligible $\varepsilon(\lambda)$ such that

$$\left|\Pr\left[\mathbf{Exp}^{\text{Obliv}}(0,\mathcal{A},\lambda,n,m_0,U)=1\right] - \Pr\left[\mathbf{Exp}^{\text{Obliv}}(1,\mathcal{A},\lambda,n,m_0,U)=1\right]\right| \geq \varepsilon(\lambda).$$

We now construct an adversary $\mathcal{B}$ that breaks the game $\mathbf{Exp}^{\text{2CPA}}$, as defined in Lemma 1, for El Gamal.

First we argue that $\mathcal{A}$ cannot distinguish based solely on observing the pattern of data blocks touched within the Circuit ORAM structure. As noted by Wang, Chan, and Shi [46], each access consists first of accessing a uniformly random path independent from all previous accesses, followed by eviction along two paths chosen independently from the access. Thus $\mathcal{A}$ can only hope to

distinguish in this manner by forcing the stash to overflow. Wang, Chan, and Shi additionally note that the probability of stash overflow is negligible in the size of the stash even for a worst-case access pattern. Therefore $\mathcal{A}$ gains at most negligible advantage by observing the Circuit ORAM access structure.

This means that $\mathcal{A}$ must either break the semantic security of El Gamal or the zero-knowledge property of an NIZK. We now assume that $\mathcal{A}$ will make at most $p(\lambda)$ queries the PVORM oracle for some polynomial p. Using this, we construct a series of hybrid distributions $H_0, \ldots, H_{p(\lambda)+1}$ modifying how the $\mathbf{Exp}^{\text{Obliv}}$ oracle works.

In hybrid H_0, the $\mathcal{O}$ operates exactly as $\mathcal{O}_{1,\mathsf{sk},C}$. In H_1, $\mathcal{O}$ operates the same way except it leverages the fact that we are in the ROM to forge all NIZKs. For H_i with $i \geq 1$, on input (u_0, u_1) from $\mathcal{A}$, the oracle applies update u_1 as in H_1 for the first $i-1$ queries, after which it applies u_0 instead. Though this may result in invalid updates, the new oracle does not check the validity and applies the update anyway with forged proofs. Because the proofs are forged, it will always succeeded in making this (forged) update. Since, by the definition of the game, $\mathcal{A}$ could not rely on submitting invalid updates in order to distinguish, this cannot improve the advantage at all.

Because we are working in the ROM and all NIZKs are GSPs, $\mathcal{A}$ receives the same view in H_0 and H_1. Whenever the PVORM oracle needs to generate a proof, it first picks a random challenge c and a response. It then computes the commitment com to ensure that the tuple is from the correct distribution, and modifies $\mathcal{A}$'s random oracle so that it receives c when querying that oracle on com. As long as the random oracle has not previously been queried on com, this strategy will work and produce exactly the same distribution as in H_1.

If there is a collision—the random oracle has been queried on com—then the experiment H_1 simply aborts. Fortunately this happens with negligible probability. Specifically, $\mathcal{A}$ makes at most $q(\lambda)$ independent queries to its random oracle for some polynomial q, and $\mathcal{O}$ must forge some constant k number of proofs for each PVORM update. This bounds the probability of collision to $\nu(\lambda) = \frac{k \cdot p(\lambda) + q(\lambda)}{2^\lambda}$, a negligible function.

We can apply the same argument to $H_{p(\lambda)+1}$ and the (unnamed) hybrid that corresponds to $\mathcal{O}_{0,\mathsf{sk},C}$ with real proofs. Thus $\mathcal{A}$ can distinguish between H_1 and $H_{p(\lambda)+1}$ with advantage at least $\varepsilon(\lambda) - 2\nu(\lambda)$. So by a standard hybrid argument, there must be some $i \in [1, p(\lambda)]$ such that $\mathcal{A}$ can distinguish between H_i and H_{i+1} with advantage at least $\frac{\varepsilon(\lambda) - 2\nu(\lambda)}{p(\lambda)}$. This too is non-negligible. For simplicity, we will denote this advantage $\varepsilon'(\lambda)$.

Next we remember that the secret key is only used to generate NIZKs in Update, meaning an adversary with only the public key can run $\mathcal{A}$ with an oracle that generates any of $H_1, \ldots, H_{p(\lambda)+1}$. $\mathcal{B}$ is exactly such an adversary.

On input $(1^\lambda, \mathsf{pk})$, $\mathcal{B}$ first guesses a uniformly random $i \in [1, p(\lambda)]$ and then runs $\mathcal{A}$. $\mathcal{B}$ then handle's $\mathcal{A}$'s PVORM oracle queries as follows. For the first $i-1$ queries (u_0, u_1), $\mathcal{B}$ applies u_1 with forged proofs—as in both H_i and H_{i+1}. Because Update uses sk only for proofs and $\mathcal{B}$ is forging proofs, it can perform the rest of Update properly with only sk. Recall that an update u consists of two plaintexts: an account ID id and a transaction value $\$v$. So to generate its chosen plaintext pairs, $\mathcal{B}$ outputs the updates specified for $\mathcal{A}$'s ith PVORM oracle query. Upon receiving a challenge pair of ciphertexts $e = (c_{\mathsf{id}}, c_v)$, $\mathcal{B}$ performs the rest of Update using that update ciphertext (and forging proofs). For all future PVORM oracle queries after the ith, $\mathcal{B}$ uses update request u_0—as in both H_i and H_{i+1}. When $\mathcal{A}$ terminates with an output, $\mathcal{B}$ outputs the same value.

We now claim that $\mathcal{B}$ has non-negligible advantage in the $\mathbf{Exp}^{\text{2CPA}}$ experiment defined above. With probability at least $\frac{1}{p(\lambda)}$, $\mathcal{B}$ will pick some i where $\mathcal{A}$ has non-negligible advantage $\varepsilon'(\lambda)$ distinguishing between H_i and H_{i+1}. If $\mathcal{B}$ receives a challenge encryption of u_1, then $\mathcal{A}$ is playing exactly the game in H_i. Similarly, if $\mathcal{B}$ is challenged with an encryption of u_0, then $\mathcal{A}$ sees exactly distribution H_{i+1}. In either case $\mathcal{B}$ will output the correct value exactly when $\mathcal{A}$ does. This means that $\mathcal{B}$ must succeed with advantage at least $\frac{\varepsilon'(\lambda)}{p(\lambda)}$, which is non-negligible.

By assumption we are working with a DDH-hard group and using El Gamal as our cryptosystem. Thus our cryptosystem is CPA secure, so by Lemma 1 no such $\mathcal{B}$ exists. This contradicts our assumption that $\mathcal{A}$ exists and therefore Construction 1 must be an oblivious PVORM. □

Theorem 4 (PVORM Public Verifiability)**.** *Construction 1 is* publicly verifiable *in the ROM.*

Proof. This result follows directly from the fact that our Update specification includes a proof of every operation as well as a range proof. By definition Ver simply verifies all NIZKs produced by Update. Therefore, if an adversary were able to fool Ver, it must be able to forge (at least) one of the proofs produced by Update.

Assume for contradiction that there exists some PPT adversary adversary $\mathcal{A}$ and non-negligible $\varepsilon(\lambda)$ such that

$$\Pr\left[\mathbf{Exp}^{\text{PubVer}}(\mathcal{A}, \lambda, n)\right] \geq \varepsilon(\lambda).$$

We note that Update produces three types of proofs. Thus we construct three new PPT adversaries $\mathcal{B}_R$, $\mathcal{B}_E$, and $\mathcal{B}_S$ that attempt to forge range proofs, proofs of plaintext equivalence on El Gamal ciphertexts, and proofs of

correct El Gamal swaps, respectively. They operate as follows.

- $\mathcal{B}_R$: On input $(\mathsf{pk}, \mathsf{sk})$, $\mathcal{B}_R$ runs $\mathcal{A}$ and outputs the resulting range proof with associated ciphertexts.
- $\mathcal{B}_E$: On input $(\mathsf{pk}, \mathsf{sk})$, $\mathcal{B}_E$ runs $\mathcal{A}$ and outputs the resulting plaintext equivalence proof and associated ciphertexts.
- $\mathcal{B}_S$: On input $(\mathsf{pk}, \mathsf{sk})$, $\mathcal{B}_S$ runs $\mathcal{A}$, picks a uniformly random El Gamal swap proof from the output, and outputs that proof and the associated ciphertexts.

Whenever $\mathcal{A}$ forges the one range proof or the one plaintext equivalence proof, $\mathcal{B}_R$ or $\mathcal{B}_E$ succeed, respectively. For $\mathcal{B}_S$, the number of El Gamal swaps executed by Update is fixed for a given PVORM configuration (tree depth, bucket size, and stash size), so if $\mathcal{A}$ forges any El Gamal swap correctness proof, $\mathcal{B}_S$ will succeed with constant probability.

By inspection of the specification of Update and a standard hybrid argument, $\mathcal{A}$ must succeed in forging at least one type of proof with non-negligible probability, hence one $\mathcal{B}_R$, $\mathcal{B}_E$, and $\mathcal{B}_S$ must succeed with non-negligible probability. As we describe in Appendix A, prior work shows that each of the associated proofs have negligible soundness error in the ROM. Thus no such adversary $\mathcal{A}$ can exist so the Solidus PVORM construction is publicly verifiable in the ROM. □

C Solidus Security Proof

In order to prove the security of Solidus, we prove that we can securely simulate $\mathcal{F}_{\text{Sol}}$ in a hybrid world with a trusted initializer, an ideal ledger, and real-world application layer. We describe the trusted initializer in Figure 9 and the ideal ledger $\mathcal{F}_{\text{Ledger}}$ in Figure 10. For bank-level signatures we employ Schnorr signatures [14, 42] which we denote by $(\mathsf{sGen}, \mathsf{Sign}, \mathsf{sVer})$.

We assume several simple pieces of behavior not directly specified by the protocol. First, each honest bank will have only one pending transaction at a time. That means that it will not approve a request (as sending or receiving bank) while there is another transaction it has approved that has not yet cleared. In the $\mathcal{F}_{\text{Ledger}}$-hybrid world, this is codified within $\mathbf{Prot}_{\text{Sol}}$, but we simply assume this property in the ideal world. Second, we assume that an honest bank will reply immediately upon receiving a transaction approval request. It may approve or abort the transaction, but it will reply in some fashion. Note that an honest bank may abort a transaction it has already approved in order to maintain availability. Finally, we assume that for an honest bank, whenever an assertion fails, the bank acts exactly as if the message it failed to process was never received.

$\mathcal{F}_{\text{Ledger}}$ uses the VerTxn function to verify full transactions. A transaction posted to $\mathcal{F}_{\text{Ledger}}$ contains

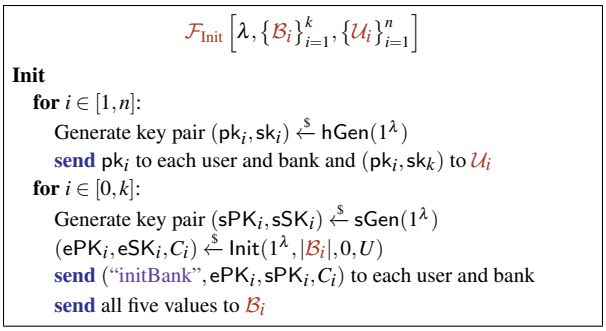

Figure 9: Ideal functionality for Solidus initialization with banks $\{\mathcal{B}_i\}$ and users $\{\mathcal{U}_i\}$.

a transaction request, a PVORM update from each bank, and a set of other proofs as described in Figure 11. Verifying all proofs requires this information as well as the previous PVORM state of each bank. Thus $\mathsf{VerTxn}(\mathcal{B}_s, \mathcal{B}_r, \mathsf{txn}, C_s, C_r)$ performs this verification given prior PVORM states C_s and C_r and returns the result. Note that we use the notation $\mathsf{LEDGER}[\mathcal{B}_s, \mathcal{B}_r]$ to denote retrieving the most recent PVORM states for $\mathcal{B}_s$ and $\mathcal{B}_r$ according to the current state of LEDGER.

We do not describe in detail how to implement $\mathcal{F}_{\text{Init}}$ or $\mathcal{F}_{\text{Ledger}}$. Instead we note that $\mathcal{F}_{\text{Init}}$ can be realized relatively simply using an existing PKI system. Once public keys are distributed, it remains only to distribute initial PVORM states with associated proofs of correct initialization.

To realize $\mathcal{F}_{\text{Ledger}}$, it is simple to modify a variety of existing consensus algorithms to perform the necessary validation. For a centralized ledger, verification is trivial. For distributed consensus, when a node receives a proposed transaction, it verifies all proofs and ignores the message if they fail to verify. For byzantine fault tolerant algorithm, this is just a malformed message which the consensus can already handle. Consensus algorithms that cannot tolerate byzantine faults should only be used when banks will not submit invalid transactions, in which case this verification has no affect.

For simplicity, we omit asset notaries from our proof. Adding them requires only small modification. Initialization must publicly distribute asset notary identities, $\mathcal{F}_{\text{Ledger}}$ must check for valid asset notary signatures, and $\mathbf{Prot}_{\text{Sol}}$ must properly reveal asset notary identities.

Figure 11 describes the $\mathcal{F}_{\text{Ledger}}$-hybrid protocol $\mathbf{Prot}_{\text{Sol}}$ for Solidus. We prove security of this protocol by constructing a simulator for it in the $\mathcal{F}_{\text{Sol}}$ world.

Theorem 1. *The Solidus protocol* $\mathbf{Prot}_{Sol}$ *satisfies Definition 1 assuming a DDH-hard group in the ROM.*

Proof. We prove that $\mathsf{Ideal}_{\mathcal{S},\mathcal{Z}}(\lambda)$ and $\mathsf{Hybrid}_{\mathcal{A},\mathcal{Z}}(\lambda)$ are indistinguishable using a sequence of hybrids. In the following, a probability is *negligible* if it is a negligible

$\mathcal{F}_{\text{Ledger}}\left[\{\mathcal{B}_i\}_{i=1}^{k}, \{\mathcal{U}_i\}_{i=1}^{n}\right]$

On receive ("approveRecvTxn", txid, txn):
- **assert** txid $\notin$ TXID
- Parse txn $\rightarrow (\mathcal{B}_s, \mathcal{B}_r, \text{txdata}_s, \sigma_s, \text{txdata}_r, \sigma_r)$
- **assert** $\text{sVer}(\mathcal{B}_s, \text{txdata}_s, \sigma_s)$
  $\wedge\ \text{sVer}(\mathcal{B}_r, \text{txdata}_r, \sigma_r)$
  $\wedge\ \text{VerTxn}(\mathcal{B}_s, \mathcal{B}_r, \text{txn}, \text{LEDGER}[\mathcal{B}_s, \mathcal{B}_r])$
- 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 $\rightarrow$ (txid, $(C, e, proof), pf^{\star}$)
- **assert** txid $\notin$ TXID
- **assert** $\text{Ver}(\text{ePK}, \text{LEDGER}[\mathcal{B}], C, e, proof)$
- **assert** $pf^{\star}$ proves e is a no-op
- TXID $\leftarrow$ TXID $\cup$ {txid}
- LEDGER $\leftarrow$ LEDGER $\|$ (abort)
- **broadcast** ("abortTxn", abort) to all banks

Figure 10: Ideal functionality for the Solidus ledger with banks $\{\mathcal{B}_i\}$ and users $\{\mathcal{U}_i\}$.

function of the security parameter λ.

We define hybrids $H_0, \ldots, H_7$. H_0 is the $\mathcal{F}_{\text{Ledger}}$-hybrid world with $\mathcal{S}$ being a "dummy" simulator that passes all messages through unchanged. H_1 allows $\mathcal{S}$ to simulate $\mathcal{F}_{\text{Ledger}}$. H_2 replaces all proofs generated by honest parties with forgeries and H_3 to replaces the contents of requests and PVORMs with arbitrary values. In H_4 $\mathcal{S}$ simulates the trusted initializer and controls all keys. H_5 isolates $\mathcal{A}$'s set of transaction IDs and H_6 drops any invalid messages from $\mathcal{A}$. Finally H_7 is equivalent to an ideal execution.

Hybrid H_0 contains a dummy simulator that passes messages between $\mathcal{A}$ and honest parties unchanged. This is identical to the $\mathcal{F}_{\text{Ledger}}$-hybrid world.

Hybrid H_1 is the same as H_0 except that $\mathcal{S}$ maintains its own simulated copy of $\mathcal{F}_{\text{Ledger}}$ that behaves as $\mathcal{F}_{\text{Ledger}}$ except for the initialization, which it does not emulate. During initialization, $\mathcal{S}$ passes the actual values sent by $\mathcal{F}_{\text{Ledger}}$ to $\mathcal{A}$ without modification. All other operations are emulated faithfully. We note that all non-initialization operations require only public information (including public keys). When an honest bank posts to $\mathcal{F}_{\text{Ledger}}$, $\mathcal{S}$ copies the message to its own copy, and when $\mathcal{A}$ posts to $\mathcal{F}_{\text{Ledger}}$, $\mathcal{S}$ first simulates the behavior on its copy, and if the post is accepted, it forwards the post to the real $\mathcal{F}_{\text{Ledger}}$.

Since all posts to $\mathcal{F}_{\text{Ledger}}$ are either dropped silently or broadcast in their entirety to all banks, $\mathcal{S}$'s faithful simulation of a copy will result in a view that is identical to real execution.

Hybrid H_2 proceeds as in H_1 except whenever $\mathcal{S}$ receives any proofs or signatures constructed by an honest party—as part of a request, PVORM update, or "postTxn" message from $\mathcal{F}_{\text{Ledger}}$—it stores the real proofs and signatures and replaces them with forgeries. $\mathcal{S}$ sends the forgeries to $\mathcal{A}$ (or the simulated $\mathcal{F}_{\text{Ledger}}$), and if a message containing those proofs would be sent back to an honest party (or forwarded to the real $\mathcal{F}_{\text{Ledger}}$), $\mathcal{S}$ puts the original (real) proofs and signatures back in place.

Note that this forgery and replacement only applies to the specific proofs and signatures constructed by honest parties. Messages from honest parties containing proofs and signatures from $\mathcal{A}$-controlled parties—such as the request signature from an $\mathcal{A}$-controlled user at an honest bank included with the final transaction—have only the honest signatures and proofs replaced. The values computed by $\mathcal{A}$ are left exactly in-tact.

As all proofs in the system are cSE NIZKs, $\mathcal{S}$ can forge proofs that $\mathcal{A}$ will accept and $\mathcal{A}$ still cannot forge proofs with non-negligible probability. Since the only thing that has changed from H_1 is these forged proofs, H_1 and H_2 are computationally indistinguishable.

Hybrid H_3 is much like H_2, but $\mathcal{S}$ also replaces the values of all encryptions generated by honest parties under honest-party keys, including PVORM values. $\mathcal{S}$ replaces these values with randomly-selected values encrypted under the same keys. Again, it saves the real values and real proofs when communicating with honest parties, but it uses the random values with $\mathcal{A}$. Since $\mathcal{S}$ only replaces values that $\mathcal{A}$ did not generate and are encrypted under public keys for which $\mathcal{A}$ does not know the secret key, the semantic security of the encryption scheme guarantees that H_3 is indistinguishable from H_2. The proofs do not present a concern as they were already forged (for the real values) in H_2, so they remain forged (for the random values) in H_3.

Hybrid H_4 differs from H_3 in that $\mathcal{S}$ now emulates the initialization in $\mathcal{F}_{\text{Init}}$. It generates fake keys and PVORMs—from the correct distribution—for all parties and sends those to $\mathcal{A}$ instead of those generated by $\mathcal{F}_{\text{Init}}$. Any encrypted values written by $\mathcal{A}$ will be encrypted under the new (fake) keys for which $\mathcal{S}$ knows the secret key, and any values intended to be read by $\mathcal{A}$ and written by an honest party will be encrypted under a key given to $\mathcal{S}$ by the real $\mathcal{F}_{\text{Init}}$. In either case, $\mathcal{S}$ can decrypt the ciphertext and re-encrypt the plaintext under the other set of keys before passing an honest message to $\mathcal{A}$ or $\mathcal{A}$'s message to an honest party. The same is true for signatures and proofs created by $\mathcal{A}$.

For encryptions under honest-party keys written by honest parties as well as proofs and signatures created by honest parties, $\mathcal{S}$ already replaced those in H_3 with random values and forgeries, respectively, so it simply does the same but under the new (fake) keys.

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

User $\mathcal{U}_i$:

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

Bank $\mathcal{B}_i$:

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

On receive ("postTxn", txid, txn) from $\mathcal{F}_{\text{Ledger}}$:
- 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** $Pend \leftarrow \perp$
- Update $B[\mathcal{B}_s] \leftarrow (\text{ePK}_s, \text{sPK}_s, C_s')$
  $B[\mathcal{B}_r] \leftarrow (\text{ePK}_r, \text{sPK}_r, C_r')$

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

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

On receive ("requestTxn", txid, $\text{ePK}_s, c_v, c_r, \sigma$) from $\mathcal{U}_s$:
- **assert** $Pend = \perp \wedge$ txid is unique
  $\wedge\ \text{hVer}(\text{ePK}_i, (\text{txid}, c_v, c_r), \sigma)$
  $\wedge$ If $\sigma = ((\alpha, \beta), pf)$, then $\text{Dec}(\text{eSK}_i, (\alpha, \beta)) = \text{pk}_s$
- Decrypt $\$v = \text{Dec}(\text{eSK}_i, c_v)$
- **assert** $0 \leq \$v \leq M_i[\mathcal{U}_s]$
- Update $(C_i', e_s, proof_s) \leftarrow \text{Update}(\text{eSK}_i, (\mathcal{U}_s, -\$v), C_i)$
- Let $c_v' = \text{Enc}(\text{ePK}_j, \$v)$
- Generate txdata_s containing:
  - $(\text{txid}, (c_v, c_r), \sigma, c_v')$
  - $(C_i', e_s, proof_s)$
  - $\text{RangePf}(e_v, t)$
  - Proof that e_s updates $\mathcal{U}_s$ by amount in c_v
  - Proof that c_v and c_v' encrypt the same value
- $Pend \leftarrow \text{txdata}_s$
- $\sigma_s = \text{Sign}(\text{sSK}_i, \text{txdata}_s)$
- **send** ("approveSendTxn", txid, $\text{txdata}_s, \sigma_s$) to $\mathcal{B}_j$

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

Figure 11: $\mathcal{F}_{\text{Ledger}}$-hybrid protocol for Solidus with banks $\{\mathcal{B}_i\}$ and users $\{\mathcal{U}_i\}$.

In this manner, all values, proofs, and signatures viewed by $\mathcal{A}$ in H_4 are the same as those in H_3, but using different encryption/signing keys and different randomness. All encryptions, proofs, and signatures generated by $\mathcal{S}$ to an honest party are similarly the same, but with different randomness. Since the keys and randomness are selected faithfully from exactly the original distributions, H_3 and H_4 are identically distributed.

Hybrid H_5 proceeds as H_4, but $\mathcal{S}$ separates the transaction IDs used by $\mathcal{A}$ from those used by honest parties. Whenever a new request comes from $\mathcal{A}$ with transaction ID $\text{txid}_{\mathcal{A}}$, $\mathcal{S}$ generates a new unique $\text{txid}_{\mathcal{F}}$ to associate with the transaction with honest parties. Whenever a message with a previously-unseen transaction ID $\text{txid}_{\mathcal{F}}$ comes in from an honest party (or $\mathcal{F}_{\text{Ledger}}$), $\mathcal{S}$ generates a new unique $\text{txid}_{\mathcal{A}}$ before forwarding to $\mathcal{A}$ (or the simulated $\mathcal{F}_{\text{Ledger}}$). If, for an incoming message in either direction, $\mathcal{S}$ has seen the ID before, there must be an associated ID in the other set, so it simply uses that.

Since only the transaction IDs have changed and the new IDs are drawn independently from the old IDs using the same methodology, H_4 and H_5 are identically distributed.

Hybrid H_6 is the same as H_5 except $\mathcal{S}$ verifies all proofs and signatures generated by $\mathcal{A}$ on all messages. If any proof or signature fails to verify, $\mathcal{S}$ drops the message and does not forward it. Because all proofs are verified in $\mathbf{Prot}_{\text{Sol}}$ (either by the receiving party or by $\mathcal{F}_{\text{Ledger}}$) before any other processing is done, and $\mathcal{Z}$ dictates that if an assertion fails, the honest party behaves as if the associated message had never arrived, this will not change any message received by $\mathcal{A}$ or the behavior of any honest parties. Similarly, H_6 drops all messages containing transaction IDs which have already been posted to $\mathcal{F}_{\text{Ledger}}$, which honest parties will similarly drop. By the simulation soundness of the NIZKs employed, $\mathcal{A}$ has a

negligible probability of forging a proof and thus there is a negligible probability of passing through a message that will be ignored anyway. Hence H_5 and H_6 are computationally indistinguishable.

Hybrid H_7 is the most complex step, as we now replace all honest-party communication with $\mathcal{F}_{\text{Sol}}$. We now describe what $\mathcal{S}$ does in H_7 whenever it would send a message to an honest party in H_6 and whenever it receives a message from $\mathcal{F}_{\text{Sol}}$ in H_7.

- When $\mathcal{S}$ would send a "requestTxn" request to an honest bank $\mathcal{B}$ on behalf of a compromised user $\mathcal{U}_s$ in H_6, $\mathcal{S}$ instead decrypts the values supplied by $\mathcal{A}$ to get the plaintext value $\$v$ and receiving user $\mathcal{U}_r$ and sends ("requestTxn", $\mathcal{U}_r$, $\$v$) to $\mathcal{F}_{\text{Sol}}$ on behalf of $\mathcal{U}_s$. Instead of creating its own $\text{txid}_{\mathcal{F}}$ to link to the $\text{txid}_{\mathcal{A}}$ for this transaction, it uses the one returned by $\mathcal{F}_{\text{Sol}}$.
- When $\mathcal{S}$ would send an "approveSendTxn" message to and honest bank in H_6, it first checks if there is an associated $\text{txid}_{\mathcal{F}}$ from $\mathcal{F}_{\text{Sol}}$, or if the message is coming unprompted from $\mathcal{A}$. In the first case it sends ("approveSendTxn", $\text{txid}_{\mathcal{F}}$) to $\mathcal{F}_{\text{Sol}}$. In the second case, it first decrypts the request included with the transaction data, which must be from a compromised user $\mathcal{U}$ at a compromised bank $\mathcal{B}$—otherwise the request would have come through $\mathcal{F}_{\text{Sol}}$ or the proofs would fail to verify and H_6 would already have dropped it. It then submits the associated "requestTxn" message to $\mathcal{F}_{\text{Sol}}$ from $\mathcal{U}$. Upon receiving an associated $\text{txid}_{\mathcal{F}}$ and ("req", $\text{txid}_{\mathcal{F}}$, $\mathcal{U}_s$, $\mathcal{B}_r$, $\$v$), $\mathcal{S}$ sends ("approveSendTxn", $\text{txid}_{\mathcal{F}}$) to $\mathcal{F}_{\text{Sol}}$.
- When $\mathcal{S}$ would send an "approveRecvTxn" message to the real $\mathcal{F}_{\text{Ledger}}$ (after passing through the simulated one), it again checks for an associated $\text{txid}_{\mathcal{F}}$ from $\mathcal{F}_{\text{Sol}}$. If none is found, then the transaction must entirely be executed by compromised entities for same reason described above. In this case, $\mathcal{S}$ decrypts the transaction details and executes the entire transaction on $\mathcal{F}_{\text{Sol}}$.

  If an $\text{txid}_{\mathcal{F}}$ is found and $\mathcal{S}$ has seen a "req" response from $\mathcal{F}_{\text{Sol}}$ but not a "aprv" message, then it must be the case that both banks are compromised. As with above, $\mathcal{S}$ finishes the transaction in order, first sending ("approveSendTxn", $\text{txid}_{\mathcal{F}}$) and then ("approveRecvTxn", $\text{txid}_{\mathcal{F}}$).

  Finally, if $\text{txid}_{\mathcal{F}}$ is found and $\mathcal{S}$ has seen a "aprv" message from $\mathcal{F}_{\text{Sol}}$ for $\text{txid}_{\mathcal{F}}$, then it simply sends ("approveRecvTxn", $\text{txid}_{\mathcal{F}}$).
- When $\mathcal{S}$ would send an "abortTxn" message to the real $\mathcal{F}_{\text{Ledger}}$, it again checks if there is an associated $\text{txid}_{\mathcal{F}}$. If there is, it sends ("abortTxn", $\text{txid}_{\mathcal{F}}$) to $\mathcal{F}_{\text{Sol}}$. If not, it generates a random txid and sends ("abortTxn", txid) to $\mathcal{F}_{\text{Sol}}$.

  Note that with negligible probability this new txid will conflict with an existing transaction ID and the abort will not be received, but except with negligible probability this will appropriately create an abort for a non-existent transaction.
- We handle $\mathcal{S}$ receiving ("req", $\text{txid}_{\mathcal{F}}$, $\mathcal{U}_s$, $\mathcal{B}_r$, $\$v$) from $\mathcal{F}_{\text{Sol}}$ in two cases.

  1. If $\mathcal{B}_r$ is honest, then $\mathcal{S}$ acts as it would in H_6 upon receiving a valid ("requestTxn", $\text{txid}_{\mathcal{F}}$, $\mathsf{ePK}_s, c_v, c_r, \sigma$) from $\mathcal{U}_s$, noting that in that case it can decrypt the identity of $\mathcal{U}_s$ and $\$v$, but not the identity of the receiving user.
  2. If $\mathcal{B}_r$ is compromised, while $\mathcal{S}$ would have forwarded a "requestTxn" message in H_6, it does not have sufficient information to create the details of that request correctly. To acquire that information, $\mathcal{S}$ immediately replies to $\mathcal{F}_{\text{Sol}}$ with ("approveSendTxn", $\text{txid}_{\mathcal{F}}$).
- When $\mathcal{S}$ receives ("aprv", $\text{txid}_{\mathcal{F}}$, $\mathcal{B}_s$, $\mathcal{U}_r$, $\$v$) from $\mathcal{F}_{\text{Sol}}$, we again have three cases.

  1. If $\mathcal{B}_s$ is compromised, then we must have been in case 2 above. Thus $\mathcal{S}$ now has sufficient information to create a complete "requestTxn" message as it would in H_6, so it does so and submits that request to $\mathcal{A}$.
  2. If $\mathcal{B}_s$ is honest but the user who originally requested this transaction $\mathcal{U}_s$ is not, then there must be some $\text{txid}_{\mathcal{A}}$ associated with $\text{txid}_{\mathcal{F}}$ and an associated request. $\mathcal{S}$ can thus manufacture an "approveSendTxn" message to submit to $\mathcal{A}$. As in H_6, $\mathcal{S}$ uses the stored request for values created by $\mathcal{U}_s$ and falsifies values created by the honest $\mathcal{B}_s$.
  3. If $\mathcal{B}_s$ and the sending user $\mathcal{U}_s$ are both honest, then $\mathcal{S}$ must create a new unique $\text{txid}_{\mathcal{A}}$ and create an "approveSendTxn" message as in H_6. Note that the values $\mathcal{S}$ could decrypt in H_6 were the identity of $\mathcal{U}_r$ and $\$v$, so it encrypts the correct values for those and falsifies other values.
- When $\mathcal{S}$ receives ("postTxn", $\text{txid}_{\mathcal{F}}$, $\mathcal{P}_s \to \mathcal{P}_r$) from $\mathcal{F}_{\text{Sol}}$, Since this proof does not handle asset notaries, we can assume $\mathcal{P}_s$ and $\mathcal{P}_r$ are both banks. There are three cases to consider.

  First we consider the simplest case: when $\mathcal{P}_r$ is a compromised bank. In this case the transaction will only clear through $\mathcal{F}_{\text{Sol}}$ after $\mathcal{S}$ successfully posts it to (the simulated) $\mathcal{F}_{\text{Ledger}}$. Thus there is nothing to do.

  Next we consider the case where $\mathcal{P}_s$ is a compromised bank but $\mathcal{P}_r$ is honest. Here $\text{txid}_{\mathcal{F}}$ must correspond to $\text{txid}_{\mathcal{A}}$ for the pending transaction in $\mathcal{S}$'s simulation

of $\mathcal{P}_r$. In order for the transaction to be approved by the sender in $\mathcal{F}_{\text{Sol}}$, $\mathcal{S}$ must have received and verified ("signRecvTxn", $\text{txid}_{\mathcal{A}}$, txdata_s) from $\mathcal{A}$. At this point $\mathcal{S}$ updates $\mathcal{P}_r$'s simulated PVORM with random values and forged proofs (as in H_6) and posts the full transaction to $\mathcal{F}_{\text{Ledger}}$. We note that $\mathcal{A}$ cannot have already submitted a transaction to $\mathcal{F}_{\text{Ledger}}$ with ID $\text{txid}_{\mathcal{A}}$ since honest banks respond instantly, so this must be in response to approving the sending of a transaction and H_6 would have dropped that message if $\text{txid}_{\mathcal{A}}$ had already been posted to $\mathcal{F}_{\text{Ledger}}$.

Finally, we consider the case where $\mathcal{P}_s$ and $\mathcal{P}_r$ are both honest. In this case $\mathcal{S}$ manufactures random updates to the respective PVORMs and forges all associated proofs. If $\text{txid}_{\mathcal{F}}$ already corresponds to some $\text{txid}_{\mathcal{A}}$, that means the requesting user was compromised, and $\mathcal{S}$ simply uses that request. Otherwise $\mathcal{S}$ selects a new unique $\text{txid}_{\mathcal{A}}$ and creates a request specification (again with random values and forged proofs). It then posts the result to the simulated $\mathcal{F}_{\text{Ledger}}$. We note that this is precisely the value that would have been posted to the simulated $\mathcal{F}_{\text{Ledger}}$ in H_6.

- When $\mathcal{S}$ receives ("abortTxn", $\text{txid}_{\mathcal{F}}$, $\mathcal{B}$) from $\mathcal{F}_{\text{Sol}}$, it first checks of $\mathcal{B}$ is compromised. If so, this must be the response after sending an abort to $\mathcal{F}_{\text{Sol}}$ and there is nothing to do. If not, $\mathcal{S}$ checks if there is a known $\text{txid}_{\mathcal{A}}$ already linked to $\text{txid}_{\mathcal{F}}$ and generates a new unique $\text{txid}_{\mathcal{A}}$ otherwise. It then generates an abort operation using random values and forged proofs, as in H_6 and posts it to $\mathcal{F}_{\text{Ledger}}$. It also clears the simulated pending transactions for $\mathcal{B}$ (which will only happen if $\text{txid}_{\mathcal{A}}$ already existed).

Thus we see that each hybrid is computationally indistinguishable from the next, H_0 corresponds to the $\mathcal{F}_{\text{Ledger}}$-hybrid world, and H_7 corresponds to the ideal world. Thus $\textbf{Prot}_{\text{Sol}}$ achieves the desired security. □

D 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.

D.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 evaluated a different construction, based on a Merkle tree, which is much more efficient for zk-SNARKs than use of Circuit ORAM.

In this zk-SNARK-friendly PVORM, each account (or other data) is stored at the leaf of a standard Merkle hash tree. The root of the tree, but no other nodes, is posted to the ledger. Upon receiving a signed update request, a bank updates one account to a valid value and modifies the Merkle tree accordingly. It then produces a zk-SNARK that it performed the update properly and that it properly verified the request's signature. The root of the new Merkle tree is the new PVORM state and the zk-SNARK is the proof.

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.

As may be seen from our results in Section 7, this construction involves proof generation times two orders of magnitude slower than those for GSPs. For single-threaded execution, or GSP PVORM with a conservative 3-bucket parameterization and 2^{15} accounts requires 0.4 seconds to generate proofs, while the zk-SNARK PVORM requires 65.5 seconds for and equivalent setup on the same machine (Amazon EC2 `c4.8xlarge` instance). For single-threaded execution, our GSP PVORM takes 0.4 sec to generate proofs with a conservative 3-bucket parameterization and 2^{15} accounts on a `c4.8xlarge` Amazon EC2 instance. Conversely, verification time for the zk-SNARK PVORM is about two orders of magnitude faster (0.0065 sec vs 0.56 sec) and proofs are quite compact (288 bytes).

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 ultimately be a valuable point in the PVORM design space.

D.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, 34, 40], 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 [26, 47]. 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 [45] 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 its implementation of SGX.

D.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 [1–3]. 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 [39]. 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 in Appendix A would require only slight modification to prove information about the known witnesses.