Solidus - Confidential Distributed Ledger Transactions via PVORM

David Lanzenberger

April 13, 2018
**Introduction**

Financial architectures:

- **Standard financial system**: transactions are between **banks**
- **Usual blockchain cryptocurrency** (e.g., Bitcoin): transactions are between **users**
Financial architectures:

- Standard financial system: transactions are between **banks**
- Usual blockchain cryptocurrency (e.g., Bitcoin): transactions are between **users**

<table>
<thead>
<tr>
<th></th>
<th>Bank</th>
<th>Bitcoin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confidentiality</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Non-Linkability</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Safety Guarantees</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Public Verifiability</td>
<td>×</td>
<td>✓</td>
</tr>
</tbody>
</table>
Financial architectures:

- Standard financial system: transactions are between **banks**
- Usual blockchain cryptocurrency (e.g., Bitcoin): transactions are between **users**

<table>
<thead>
<tr>
<th></th>
<th>Bank</th>
<th>Bitcoin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confidentiality</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Non-Linkability</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Safety Guarantees</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Public Verifiability</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>
Financial architectures:

- **Standard financial system**: transactions are between **banks**
- **Usual blockchain cryptocurrency** (e.g., Bitcoin): transactions are between **users**

<table>
<thead>
<tr>
<th></th>
<th>Bank</th>
<th>Bitcoin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confidentiality</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Non-Linkability</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Safety Guarantees</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Public Verifiability</td>
<td>×</td>
<td>✓</td>
</tr>
</tbody>
</table>
Financial architectures:

- **Standard financial system**: transactions are between **banks**
- **Usual blockchain cryptocurrency** (e.g., Bitcoin): transactions are between **users**

<table>
<thead>
<tr>
<th></th>
<th>Bank</th>
<th>Bitcoin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confidentiality</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Non-Linkability</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Safety Guarantees</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Public Verifiability</td>
<td>×</td>
<td>✓</td>
</tr>
</tbody>
</table>
Combine distributed ledger with bank-intermediated architecture:

- Confidentiality
- Non-Linkability
- Safety guarantees
- Public verifiability

Bank-intermediated blockchain systems exist already, but...
"send 5CHF to Bob"
Classical Banking

<table>
<thead>
<tr>
<th>User</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>6-5=1</td>
</tr>
<tr>
<td>C</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>User</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>0+5=5</td>
</tr>
<tr>
<td>D</td>
<td>9</td>
</tr>
</tbody>
</table>

"you received 5CHF"
Solidus - Confidential Distributed Ledger Transactions via PVORM
Trust Model & Security Goals

Trust Model:
- Availability and liveness of the ledger
- No network adversary

Safety goals:
- No user’s balance decreases without explicit permission
- Account balances are always non-negative
- Transactions are zero-sum (except...)

Confidentiality goals:
- Balances are visible only to the bank
- Non-Linkability: Cannot determine if two transactions involving the same bank involved the same account.
Solidus
Bank-Intermediated System

\[ \mathcal{U}_1^s \quad \mathcal{B}_s \quad \mathcal{U}_2^s \]

\[
\begin{array}{c|c}
\text{pk}_1^s & \$b_1^s \\
\end{array} \quad \begin{array}{c|c}
\text{pk}_2^s & \$b_2^s \\
\end{array}
\]

This figure represents a bank-intermediated system where each bank has a set of users, each with a public key and a balance. The middle bank, \( \mathcal{B}_s \), manages the transactions between \( \mathcal{U}_1^s \) and \( \mathcal{U}_2^s \).
Solidus Architecture

$T : U_2^s \rightarrow U_1^r : \$v$

**Solidus Architecture**

**Solidus**

PVORM

 Ledger

**Session C5: Using Blockchains**

**CCS'17, October 30-November 3, 2017, Dallas, TX, USA**
To prevent replay attacks, Solidus includes a Transaction ID that is valid at a specific time window. If a random bit string (e.g., a GUID) is used, verification would require the ID of every transaction over the lifetime of the system. To address this, Solidus employs Schnorr signatures [16, 55], denoted as $\text{Schnorr}$, and hidden-public-key signatures (see Appendix A.3) and banks $B_f$.

Solidus is agnostic to the ledger implementation, so we wish to employ our idealized ledger. We could instantiate the trusted initializer and an idealized ledger. We could define the Solidus protocol for simplicity, but in reality, solidus must support new accounts, so Solidus must support this. To create an account, the initializer uses a circuit ORAM to check for a set accounts, so Solidus must support this. To create an account, the initializer and an idealized ledger. We could instantiate the trusted initializer with banks $B_f$ and users $U_f$.

For a transaction from $U_s$ to $U_r$, Solidus first requires the identity of an account and replaces the ledger. This allows verification of the transaction. Note that the main Solidus protocol, $\text{Solidus}$, settles transactions after a series of operations: posting a transaction, preparing the transaction, updating the ledger, and signing the update. The protocol parties and the adversary can settle the transaction.

For simplicity, we define the Solidus protocol, $\text{Solidus}$, as follows:

1. **Request**
   - $U_s$ sends a request to the ledger

2. **Verify & Prepare**
   - $B_s$ verifies the request and prepares the transaction

3. **PVORM Update**
   - $B_s$ updates the PVORM

4. **Sign**
   - $B_s$ signs the transaction

5. **Settle**
   - Solidus settles the transaction

The protocol is designed to be flexible and secure, allowing for the efficient and confidential execution of transactions.
PVORM
Logical state of a single bank:

<table>
<thead>
<tr>
<th>Public-Key</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>$pk_A$</td>
<td>6</td>
</tr>
<tr>
<td>$pk_B$</td>
<td>9</td>
</tr>
<tr>
<td>$pk_C$</td>
<td>7</td>
</tr>
</tbody>
</table>

Balance confidentiality?
Logical state of a single bank:

<table>
<thead>
<tr>
<th>Public-Key</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>$pk_A$</td>
<td>6</td>
</tr>
<tr>
<td>$pk_B$</td>
<td>9</td>
</tr>
<tr>
<td>$pk_C$</td>
<td>7</td>
</tr>
</tbody>
</table>

Balance confidentiality? **Encryption.**
Logical state of a single bank:

<table>
<thead>
<tr>
<th>Public-Key</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>$pk_A$</td>
<td>6</td>
</tr>
<tr>
<td>$pk_B$</td>
<td>9</td>
</tr>
<tr>
<td>$pk_C$</td>
<td>7</td>
</tr>
</tbody>
</table>

Balance confidentiality? **Encryption.**

Non-Linkability?
Logical state of a single bank:

<table>
<thead>
<tr>
<th>Public-Key</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>$pk_A$</td>
<td>6</td>
</tr>
<tr>
<td>$pk_B$</td>
<td>9</td>
</tr>
<tr>
<td>$pk_C$</td>
<td>7</td>
</tr>
</tbody>
</table>

Balance confidentiality? **Encryption.**

Non-Linkability?

Public Verifiability?
$N = 7$ nodes, bucket size $B = 2$

Logical balance state $M = \{pk_A \mapsto 6, \ pk_B \mapsto 9, \ pk_C \mapsto 7\}$

$PosMap = \{pk_A \mapsto \ell_0, \ pk_B \mapsto \ell_2, \ pk_C \mapsto \ell_1\}$
$N = 7$ nodes, bucket size $B = 2$

Logical balance state $M = \{pk_A \mapsto 6, pk_B \mapsto 9, pk_C \mapsto 7\}$

$PosMap = \{pk_A \mapsto \ell_0, pk_B \mapsto \ell_2, pk_C \mapsto \ell_1\}$
Alice ($pk_A$) wants to send 5CHF to Bob ($pk_B$). So she sends to following to her bank:

$$[\text{enc}(5, pk_{\text{Bank}A}), \text{enc}(pk_B, pk_{\text{Bank}B})]_{sk_A^{-1}}$$
Alice ($pk_A$) wants to send 5CHF to Bob ($pk_B$). So she sends to following to her bank:

$$[enc(5, pk_{Bank_A}), enc(pk_B, pk_{Bank_B})]_{sk_A^{-1}}$$

How does the bank subtract 5CHF from Alice’s account?
Transaction: \[\text{enc}(5, pk_{\text{BankA}}), \text{enc}(pk_B, pk_{\text{BankB}})\]_{sk_A}^{-1}

Logical balance state \(M = \{pk_A \mapsto 6, pk_B \mapsto 9, pk_C \mapsto 7\}\)

\(\text{PosMap} = \{pk_A \mapsto \ell_0, pk_B \mapsto \ell_2, pk_C \mapsto \ell_1\}\)
Transaction: \[\operatorname{enc}(5, pk_{B_{\text{A}}}), \operatorname{enc}(pk_{B}, pk_{B_{\text{B}}})\]^{sk_{A}}^{-1}

Logical balance state \(M = \{pk_{A} \mapsto 6, pk_{B} \mapsto 9, pk_{C} \mapsto 7\}\)

\(\text{PosMap} = \{pk_{A} \mapsto \ell_{0}, pk_{B} \mapsto \ell_{2}, pk_{C} \mapsto \ell_{1}\}\)
Transaction: $[\text{enc}(5, pk_{\text{BankA}}), \text{enc}(pk_B, pk_{\text{BankB}})]_{sk_A}^{-1}$

Logical balance state $M = \{pk_A \mapsto 6, pk_B \mapsto 9, pk_C \mapsto 7\}$

$\text{PosMap} = \{pk_A \mapsto \ell_0, pk_B \mapsto \ell_2, pk_C \mapsto \ell_1\}$
Transaction: \[
[\text{enc}(5, \text{pk}_{B\text{ank}_A}), \text{enc}(\text{pk}_B, \text{pk}_{B\text{ank}_B})]_{\text{sk}_A^{-1}}
\]

Logical balance state \( M = \{ \text{pk}_A \leftrightarrow 6, \text{pk}_B \leftrightarrow 9, \text{pk}_C \leftrightarrow 7 \} \)

\( \text{PosMap} = \{ \text{pk}_A \leftrightarrow \ell_0, \text{pk}_B \leftrightarrow \ell_2, \text{pk}_C \leftrightarrow \ell_1 \} \)
Transaction: \( [\text{enc}(5, pk_{BankA}), \text{enc}(pk_B, pk_{BankB})]_{sk_A}^{-1} \)

Logical balance state \( M = \{ pk_A \mapsto 6, pk_B \mapsto 9, pk_C \mapsto 7 \} \)

\( \text{PosMap} = \{ pk_A \mapsto \ell_0, pk_B \mapsto \ell_2, pk_C \mapsto \ell_1 \} \)
Transaction: $[\text{enc}(5, pk_{\text{Bank}_A}), \text{enc}(pk_B, pk_{\text{Bank}_B})]_{sk_A^{-1}}$

Logical balance state $M = \{pk_A \mapsto 6, pk_B \mapsto 9, pk_C \mapsto 7\}$

$\text{PosMap} = \{pk_A \mapsto \ell_0, pk_B \mapsto \ell_2, pk_C \mapsto \ell_1\}$
Transaction: $[\text{enc}(5, pk_{\text{Bank}A}), \text{enc}(pk_B, pk_{\text{Bank}B})]_{sk_A^{-1}}$

Logical balance state $M = \{pk_A \mapsto 6, pk_B \mapsto 9, pk_C \mapsto 7\}$

$\text{PosMap} = \{pk_A \mapsto \ell_0, pk_B \mapsto \ell_2, pk_C \mapsto \ell_1\}$
El-Gamal.

- Cyclic group $G = \langle g \rangle$
- Message space $\mathcal{M} = G$
- $\text{enc}(m_1, pk) \cdot \text{enc}(m_2, pk) = \text{enc}(m_1 \cdot m_2, pk)$
- $\text{enc}(g^i, pk) \cdot \text{enc}(g^j, pk) = \text{enc}(g^{i+j}, pk)$
- $\text{enc}(g^i, pk) / \text{enc}(g^j, pk) = \text{enc}(g^{i-j}, pk)$
El-Gamal.

- Cyclic group $G = \langle g \rangle$
- Message space $\mathcal{M} = G$

$$
\text{enc}(m_1, pk) \cdot \text{enc}(m_2, pk) = \text{enc}(m_1 \cdot m_2, pk)
$$

$$
\text{enc}(g^i, pk) \cdot \text{enc}(g^j, pk) = \text{enc}(g^{i+j}, pk)
$$

$$
\text{enc}(g^i, pk) / \text{enc}(g^j, pk) = \text{enc}(g^{i-j}, pk)
$$

Homomorphic encryption.

$$
\text{enc}(g^6, pk_{BankA}) / \text{enc}(g^5, pk_{BankA}) = \text{enc}(g^{6-5}, pk_{BankA}) = \text{enc}(g^1, pk_{BankA})
$$
Transaction: \[ \text{enc}(5, \text{pk}_{\text{BankA}}), \text{enc}(\text{pk}_B, \text{pk}_{\text{BankB}}) \]_{sk_A^{-1}}

Logical balance state \( M = \{\text{pk}_A \mapsto 1, \text{pk}_B \mapsto 9, \text{pk}_C \mapsto 7\} \)

\( \text{PosMap} = \{\text{pk}_A \mapsto \ell_0, \text{pk}_B \mapsto \ell_2, \text{pk}_C \mapsto \ell_1\} \)
Sample **new label** for $pk_A$ u.a.r.: $\ell_{new} \leftarrow \{\ell_0, \ell_1, \ell_2, \ell_3\}, \ell_{new} = \ell_3$

Logical balance state $M = \{pk_A \mapsto 1, pk_B \mapsto 9, pk_C \mapsto 7\}$

$PosMap = \{pk_A \mapsto \ell_3, pk_B \mapsto \ell_2, pk_C \mapsto \ell_1\}$
Pick two eviction paths, say $\ell_0$ and $\ell_3$.
Logical balance state $M = \{ pk_A \mapsto 1, pk_B \mapsto 9, pk_C \mapsto 7 \}$
$PosMap = \{ pk_A \mapsto \ell_3, pk_B \mapsto \ell_2, pk_C \mapsto \ell_1 \}$

fixed: $(pk_A, 1)$

stash: $(0, 0), (0, 0), (0, 0), (0, 0)$

$(pk_C, 7), (0, 0)$

$(0, 0), (0, 0)$

$(0, 0), (0, 0)$

$(0, 0), (0, 0)$

$(0, 0), (0, 0)$
Pick two **eviction paths**, say $\ell_0$ and $\ell_3$.

Logical balance state $M = \{pk_A \mapsto 1, pk_B \mapsto 9, pk_C \mapsto 7\}$

$PosMap = \{pk_A \mapsto \ell_3, pk_B \mapsto \ell_2, pk_C \mapsto \ell_1\}$
Experiments
We measured the concrete performance of PVORM. We now present performance results for our PVORM and Solidus prototypes. Our Solidus implementation is 4300 lines of Java code, and employed the precomputation optimization (Section 6.1). These benchmarks do not include the precomputation time.

7.1 PVORM Performance

Figure 8 shows the single-threaded performance of PVORM as we vary bucket and stash sizes. As expected, larger buckets are slower and runtime grows linearly with stash size. This measures the performance-privacy trade-off.

Figure 9 shows the single-threaded performance of a fully distributed Solidus system with 2 to 12 banks. Each bank runs on its own Amazon EC2 instances having two 10-core CPUs, we present scaling to only 10 worker threads. Note that with one server per bank we expect meaningful speedup. Second, our prototype implementation does not distribute to multiple hosts and scales poorly to multi-CPU instances and maintains a PVORM with size 3 buckets, a single PVORM update with proof size 190 KB (or 114 KB if compressed). Points can be compressed to a single bit and a field element, but decompression imposes nontrivial overhead.

Despite this, memory consumption peaks at only 880 MB. Our prototype requires a complete copy of the PVORM in memory. Proof size and memory usage.

The Experiments section discusses the performance results for various configurations and levels of parallelism. A single update contains many NIZKs that can be created where all computation is parallelized with no overhead. Because the proof of each pairwise swap can be computed or verified independently, we expect performance to scale well 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 instances. Therefore, we stop at 10 for a combination of parallelized and non-parallelized operations under different configurations and levels of parallelism.

7.2 Solidus System Performance

Figure 10 shows performance for a single operation. A single update contains many NIZKs that can be created as long as the rollback can be placed after it while changes executed by T's update, thus allowing verifiers to check B = 3 V
er
B = 2 Up
date
Figure 8: PVORM imposing nontrivial overhead.
Experiments

We measured the concrete performance of our PVORM and Solidus prototypes. Our Solidus implementation is 4300 lines of Java code, and employed the precomputation optimization (Section 6.1). These benchmarks do not include the precomputation time.

Our PVORM construction supports highly parallelized implementation and only one server per bank, allowing it to scale up using multiple servers per bank to achieve meaningful speedup. Second, our prototype implementation does not distribute to multiple hosts and scales poorly to multi-CPU instances and maintains a PVORM with size of 7 buckets, capacity of 15, and stash size of 25 of size 25 stash, and capacity of

An elliptic curve point is an ordered pair of elements of $\mathbb{F}_p$ for some prime $p$. Points can be compressed by noting that the order of 2 divides the size of the group of points on the curve, so we can consider only the even part of each coordinate. Because the proof of each pairwise swap can be computed or verified independently, this imposes nontrivial overhead.

### Table 1: Solidus System Performance

<table>
<thead>
<tr>
<th>Worker Threads</th>
<th>Throughput (ops/sec) Generate</th>
<th>Throughput (ops/sec) Verify</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.37</td>
<td>0.58</td>
</tr>
<tr>
<td>4</td>
<td>0.74</td>
<td>0.86</td>
</tr>
<tr>
<td>6</td>
<td>0.93</td>
<td>0.98</td>
</tr>
<tr>
<td>8</td>
<td>1.10</td>
<td>1.14</td>
</tr>
</tbody>
</table>

### Figure 8: PVORM Performance with Capacity Scaling

Figure 8 shows the single-threaded performance of our PVORM as the capacity scales. As expected, the binary tree structure results in clearly logarithmic scaling.

### Figure 9: PVORM Capacity Scaling

Figure 9 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 stash overflow probability of around 0.3.

### Figure 10: Performance for a Single Operation

We now present performance results for our PVORM and Solidus. We ran all experiments on Amazon EC2 instances c4.8xlarge, highly limiting transactions into blocks, as is done in systems like Bitcoin. If a transaction aborts, instead of reprocessing it, we can alleviate some of this performance penalty by bundling several transactions together and committing them to the blockchain in one transaction. This rollback must provably revert any changes executed by a transaction later in the same block. This rollback can include a rollback operation under different configurations and levels of parallelism.

PVORM operations are CPU-bound, so adding threads beyond the number of CPU cores produces no meaningful speedup.
Experiments

We measured the concrete performance of our PVORM and Solidus with the stash size. As bucket and stash sizes determine the chance of stash overflow, Figure 9 shows the single-threaded performance of our PVORM as the capacity scales. As expected, the binary tree recompute operation, which constitutes the PVORM, we use BouncyCastle for our prototype. Our Solidus implementation is 4300 lines of Java code, but memory consumption peaks at only 880 MB. Despite this, memory consumption peaks at only 880 MB. An elliptic curve point is an ordered pair of elements of \( \mathbb{F}_p \). Points can be compressed as 100.

We expect performance to scale well beyond 10 threads. The experiments show that the number of worker threads affects the throughput significantly. In the perfect scaling case, the throughput increases linearly with the number of threads. However, in real-world scenarios, the throughput increases at a rate lower than perfect scaling, indicating some overhead.

In our experiments, we observed that the throughput for updates is higher than for verifications. This is expected because updates require more cryptographic operations than verifications. The graph shows that the throughput for updates and verifications increases with the number of threads, but not as much as the perfect scaling line. This suggests that the overhead due to communication and coordination among threads limits the improvement in throughput.

For a PVORM with size 3 buckets, a size 25 stash, and capacity of 2\(^3\), we present scaling to only 10 worker threads. Note that with 10 threads, the throughput is significantly lower than with 2 threads. This is because the overhead due to communication and coordination among threads limits the improvement in throughput. In practice, our prototype requires a complete copy of the PVORM in memory, which is effectively parallelized on the same CPU. This likely explains some of the reduced scaling in that case.

The PVORM performance results for our PVORM and Solidus are shown in Figure 9. The results show that our PVORM construction supports highly parallelized transaction updating. Solidus is designed to be highly parallelized, and our parallelism is effective in practice. We ran all experiments on c4.8xlarge Amazon EC2 instances, which have two 10-core CPUs, and only one server per bank. This was never processed at all. There is, however, no need to compute or verify all of these transactions into blocks, as is done in systems like Bitcoin. If transactions are not distributed to multiple hosts and scales poorly to multi-CPU architectures. Since one coordination thread, which does very little work, is exactly one coordination thread. This is likely explained by the fact that the proof of each pairwise swap can be computed or verified independently, we expect performance to scale well beyond 10 threads.
"We expect performance to scale well beyond 10 threads – possibly as high as 100."
Experiments

![Graph showing throughput vs. number of banks]

**Solidus - Confidential Distributed Ledger Transactions via PVORM**

We finally compare our prototype's performance to that of a PVORM using a compact Merkle tree structure. Each account is stored at the leave of a standard Merkle hash tree, the root of which is posted to the ledger. To update the PVORM, a bank updates one account by Chaum's protocol [22, 23] and refined in a long series of works, e.g., [15, 17, 18, 19].

Alternative schemes such as Monero [3], a relatively popular anonymous cryptocurrency. A number have been proposed and deployed, including Hawk [37], providing partial transaction-graph concealment. Serious weaknesses in Monero's anonymity have recently been identified, however [47], while MimbleWimble [35], a pseudonymous protocol, offers enhance privacy for this purpose [31]. In these schemes, trust is centralized. A single authority issues digital currency, which is spent and redeems coins that are anonymized using blind signatures or zero-knowledge proofs.

Reliable cryptocurrencies [13] use zk-SNARKs to ensure conservation of money, prevent double spending, and hide both transaction values and the system’s transaction graph. Consequently, unlike Solidus, it requires trusted setup, which in practice must be centralized (as multiparty computation on a large scale is impractical). This is the main disadvantage of zk-SNARKs, as their setup is expensive and increases verification time. However, the advantage of using zk-SNARKs is that they are extremely fast, even highly parallel proof generation is more than two orders of magnitude faster than the GSP PVORM on a c4.8xlarge EC2 instance. While verification is extremely fast, even highly parallel proof generation is more than two orders of magnitude faster than the GSP PVORM. For this to improve overall system throughput, the system would need to verify every proof to the ledger. To update the PVORM, a bank updates one account by Chaum's protocol [22, 23] and refined in a long series of works, e.g., [15, 17, 18, 19].

Mixes partially obscure the transaction graph in an existing ledger. For example, mix networks rely on zk-SNARKs to ensure conservation of money, prevent double spending, and hide transaction values and the system’s transaction graph. Consequently, unlike Solidus, it requires centralized setup. A single authority issues digital currency, which is spent and redeems coins that are anonymized using blind signatures or zero-knowledge proofs. The advantage of using zk-SNARKs is that they are extremely fast, even highly parallel proof generation is more than two orders of magnitude faster than the GSP PVORM. For this to improve overall system throughput, the system would need to verify every proof to the ledger. To update the PVORM, a bank updates one account by Chaum's protocol [22, 23] and refined in a long series of works, e.g., [15, 17, 18, 19].

**Table 1: Performance of PVORM using zk-SNARKs.**

<table>
<thead>
<tr>
<th>Number of Threads</th>
<th>Proof Size (GB)</th>
<th>Verification Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>45</td>
<td>1.7</td>
</tr>
<tr>
<td>2</td>
<td>53</td>
<td>2.5</td>
</tr>
<tr>
<td>4</td>
<td>76</td>
<td>5.3</td>
</tr>
</tbody>
</table>

In our exploration of a zk-SNARK variant of Solidus in Section 7.3, we use a compact Merkle tree structure. Each account is stored at the leave of a standard Merkle hash tree, the root of which is posted to the ledger. To update the PVORM, a bank updates one account by Chaum's protocol [22, 23] and refined in a long series of works, e.g., [15, 17, 18, 19].

Mixed［57］. As mixes' costs are extremely high, the cost normally associated with them makes them impractical for widespread use. The advantage of using zk-SNARKs is that they are extremely fast, even highly parallel proof generation is more than two orders of magnitude faster than the GSP PVORM. For this to improve overall system throughput, the system would need to verify every proof to the ledger. To update the PVORM, a bank updates one account by Chaum's protocol [22, 23] and refined in a long series of works, e.g., [15, 17, 18, 19].
First bank-intermediated ledger-based system with:
- Balance confidentiality
- Non-Linkability
- Safety guarantees
- Public Verifiability

Possible improvements:
- Theoretical
- Implementation
Questions?