Proving the correct execution of concurrent services in zero-knowledge

Srinath Setty, Sebastian Angel, Trinabh Gupta, and Jonathan Lee
Microsoft Research, UPenn, UCSB
Software verification

Verifies that code obeys a desired specification (first three talks)

Proving correct executions

A cryptographic proof that desired code was correctly executed (this talk)

Neither subsumes the other
Consider a cloud-hosted wallet service (e.g., Square, WeChat Pay)

API
- Issue (…)
- Transfer(…)
- Withdraw(…)

key-value store

<table>
<thead>
<tr>
<th>clientId</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>200</td>
</tr>
<tr>
<td>2</td>
<td>500</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
</tr>
</tbody>
</table>

Verify trace by replay
Issues with verifiability via record-and-replay

1. Sacrifices privacy: exposes requests and the internal state to a verifier
   • For example: account balances in the wallet app

2. Verification via replay is expensive
   • Verifiers must reexecute all requests
   • Recorded trace can be large ➔ network costs are high

Verifiable state machines address both problems
A verifiable state machine:

- Proofs are zero-knowledge: they do not reveal requests, responses, or the state.
- Proofs are succinct: each proof is small and verification is inexpensive.
- If the service errs, verifiers output reject (except for a small probability, $<1/2^{128}$).

<table>
<thead>
<tr>
<th>clientId</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>200</td>
</tr>
<tr>
<td>2</td>
<td>500</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
</tr>
</tbody>
</table>

Verifiable state machine:

- Verifiers output reject in the case of a service error (except with a probability of $<1/2^{128}$).
Prior work on verifiable state machines

- The underlying theory dates back to 90s: Babai et al. [STOC91], ...
  \[ \text{cost reductions by } 10^{20}\times \]

- **Pepper** [HotOS11, NDSS12], **CMT** [ITCS12], **Ginger** [Security12], **TRMP** [HotCloud12]

- **Zaatar** [EuroSys13], **Pinocchio** [S&P13], **Allspice** [S&P13], **SNARKs-for-C** [CRYPTO13]

- **Pantry** [SOSP13], **Geppetto** [S&P15], **CTV** [EUROCRYPT15], **vSQL** [S&P17], ...

  \[ \text{support stateful computations} \]

  \[ \text{storage interfaces: key-value stores, etc.} \]
Prior work suffers from two major issues

1. Producing proofs about storage operations is computationally expensive
   Several seconds to minutes of CPU-time/operation

2. They can only produce proofs about sequential executions → each request must be processed before the next

For the wallet service app (on a single CPU core):
   Pantry [SOSP13] achieves < 0.15 requests/second
   Geppetto [S&P15] achieves < 0.002 requests/second
Our system: Spice

• Features a new storage primitive: 29—2000x more efficient
• Supports concurrent request processing, with transactional semantics
• Includes a toolchain:

- We built three apps: a wallet service, payment network, and a dark pool
- Throughput: 488—1048 reqs/sec (512 CPU-cores)
  • This is 18,000—685,000 higher throughput than prior work
Rest of this talk

• Background

• Overview of Spice

• Experimental results
Background: Pantry \cite{SOSP13}

Under Pantry, a service is expressed using:

- Arithmetic operations
- Bitwise operations
- Conditional control flow
- Volatile memory (with pointers)
- Loops (with bounded iterations)

subset of C + storage primitives

- Key-value ops: get, put, etc.
Mechanics of Pantry [SOSP13] to produce proofs

C program

```c
int f(int a, int b) {
    return a*2 - b;
}
```

compile
(translates C to constraints)

```
x_1 - input a = 0
x_2 - input b = 0
(x_1 \cdot 2) - x_3 = 0
(x_3 - x_2) - x_4 = 0
x_4 - output = 0
```

execute and produce proofs
(an argument protocol)

verifier

service

proof

build
Background: Pantry[\textit{SOSP13}]

Under Pantry, a service is expressed using:

- Arithmetic operations
- Bitwise operations
- Conditional control flow
- Volatile memory (with pointers)
- Loops (with bounded iterations)
An attempt:

```java
Value get(Key k) {
    Value v = service_get(k);
    return v;
}
```

<table>
<thead>
<tr>
<th>clientId</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>200</td>
</tr>
<tr>
<td>2</td>
<td>500</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
</tr>
</tbody>
</table>

service supplies state

key-value store maintained by service

service could supply incorrect values
Merkle trees provide the necessary building block

```
Value get(Key k, Root R) {
    Value *v = service_get(k);
    assert R' == R; // fails for incorrect value
    return v[0];
}
```
Issues with using Merkle trees for key-value stores

1. Cost of a get/put is logarithmic in the size of the state

2. The root of the tree serves as a contention point
   → supports only sequential executions
Rest of this talk

• Background

• Overview of Spice

• Experimental results
Spice in a nutshell:

- Arithmetic and bitwise ops
- Conditionals, loops, memory
- ...

subset of C + storage primitives

Compile and apply argument protocol

Succinct zero-knowledge proof

[Blum et al. FOCS91, Clarke et al. ASIACRYPT03, Arasu et al. SIGMOD17]
service’s state

read-set
write-set

Insert\((k,v)\):

get\((k)\) $\rightarrow$ \((k,v,0)\)

get\((k)\) $\rightarrow$ \((k,v,0)\), \((k,w,1)\)

\((k,w,0)\)

\((k,v,0)\), \((k,w,1)\)

\((k,v,0)\)

\((k,v,0)\), \((k,v,1)\)

read-set is a not subset of write-set

read-set is a subset of write-set
An equivalent of Merkle root

```c
struct set-root {
    set-hash rs; // set-hash of read-set
    set-hash ws;
}
```

Service’s state

\[
\text{Set-Hash}\left(\begin{array}{c}
A, \\
B
\end{array}\right) = \text{Set-Hash}\left(\begin{array}{c}
A
\end{array}\right) \times \text{Set-Hash}\left(\begin{array}{c}
B
\end{array}\right) = \text{Set-Hash}\left(\begin{array}{c}
B
\end{array}\right) \times \text{Set-Hash}\left(\begin{array}{c}
A
\end{array}\right)
\]
Takeaways on set-based storage:

- get, put add an element to read-set and write-set
- service *periodically proves* read-set \(\subseteq\) write-set
- non-conflicting set operations commute
- cost of a get, put is a constant
- cost is linear in state size, but amortized over a batch
- multiple writers and concurrent request processing
We built transactions and apps atop set-based storage

- **Wallet service**
- **Payment network**
- **Dark pool**

**Transactions**: one-shot, 2PL, ...

**Mutual-exclusion**: lock, unlock

**Set-based key-value store**

**Apps** (1,500 LOC)

**C library** (2,000 LOC)
Rest of this talk

• Background

• Overview of Spice

• Experimental results
Evaluation questions

1. How does Spice compare with the prior state-of-the-art?
2. What is the end-to-end performance of apps built with Spice?

Evaluation testbed:

Azure D64s_v3 instances: 32 CPUs, 2.4 Ghz Intel Xeon, 256 GB RAM, running Ubuntu 17.04
(1) How does Spice compare to prior work?

A million key-value pairs
Transactions with a single operation, keys chosen with a uniform distribution
Metric: number of ops/second (i.e., proofs/sec)

<table>
<thead>
<tr>
<th></th>
<th>get</th>
<th>put</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pantry [SOSP13]</td>
<td>0.078</td>
<td>0.039</td>
</tr>
<tr>
<td>Pantry++</td>
<td>0.15</td>
<td>0.076</td>
</tr>
<tr>
<td>Geppetto [S&amp;P15]</td>
<td>0.002</td>
<td>0.002</td>
</tr>
<tr>
<td>Spice (1-thread)</td>
<td>3.6</td>
<td>3.6</td>
</tr>
<tr>
<td>Spice (512-threads)</td>
<td>1,366</td>
<td>1,370</td>
</tr>
</tbody>
</table>
(2) End-to-end performance with varying #CPUs

- TPS is 18,000—685,000x better than prior state-of-the-art
- Verification throughput: >1,000 proof verifications/sec (4 CPU-cores)
Limitations of Spice

1. CPU-cost to produce proofs remains large (compared to executions without proofs): $>1000\times$

2. Spice amortizes the cost of producing a proof (that read-set subset write-set) over a batch of requests
   • Introduces latency for producing proofs (7.5 minutes in our experiments)
   • Tunable, but lower latency increases CPU-costs
Summary

• Verifiable state machines add verifiability to services—without compromising their privacy

• Spice is a substantial progress toward building verifiable state machines
  • 18,000—685,000x better performance (over prior state-of-the-art)
  • Spice supports realistic apps with thousands of transactions/sec

• We predict: Spice or a variant will be a key tool in building secure systems