Increasing Throughput
with Sharded Blockchains

Blockchains Study Hall
Scaling blockchains
Scaling blockchains
High throughput
BFT resilience
Fast finality
Linear scalability
Scalability

The more machines you have, the bigger your throughput
State Sharding
State Sharding

shard 1

shard 2

shard 3

User

Data Shards
State Sharding
State Sharding

o1, o3

o6, o7

o4, o8

o6, o7
State Sharding
An example transaction

\[ T(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]
State Sharding
An example transaction

\[ T(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]
State Sharding

Only two acceptable final states

X₁
Shard 1

X₂
Shard 2

Y₁
Shard 1

Y₂
Shard 2

Y₃
Shard 3

Shard 1

Shard 2

Shard 3
Cross-Shard Consensus
How do shards communicate with each other?
Mutex-Based Protocols

\[ T(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]

- client
- shard 1
- shard 2
- shard 3
Mutex-Based Protocols

\[ T(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]
Mutex-Based Protocols

\[ T(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]
Mutex-Based Protocols

\[ T(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]
Mutex-Based Protocols

\[ T(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]
Mutex-Based Protocols

Choose the assumptions

- Synchrony (for safety): Add an expiration to the receipt
- Infinite memory: Keep receipts forever
When BFT fails, auditing mechanisms are in place to trace malicious parties. We integrate a value system, named CSCoin, that supports user-defined smart contracts. Chainspace aims to be an "open" system: it allows anyone to shard state and execute user-supplied transactions on their objects. The system is scalable, by sharding state and the execution of transactions, and using certificate verification, Chainspace offers extensibility through Byzantine Fault Tolerance (BFT), and extremely high-performance of the system about its scaling and other features; we illustrate a section on a real distributed set of nodes and under varying transaction loads.

The contracts for privacy-friendly smart-metering and distinction is key to supporting privacy-friendly smart contracts and evaluates their performance. It presents Chainspace, a system that can scale arbitrarily as the number of nodes increase, tolerates byzantine failures, and can be fully and publicly audited. The system is scalable, by sharding state and the execution of transactions, and using certificate verification, Chainspace offers extensibility through Byzantine Fault Tolerance (BFT), and extremely high-performance of the system about its scaling and other features; we illustrate a section on a real distributed set of nodes and under varying transaction loads.
Cross-Shard Consensus

Byzantine Agreement + 2-Phases Atomic Commit
Spoiler alert: Insecure under parallel composition
S-BAC

\[ T(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]

- Client
- Shard 1
- Shard 2
- Shard 3
S-BAC

\[ T(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]
S-BAC

\[ T(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]
S-BAC

\[ T(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]
S-BAC

\[ T(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]
S-BAC

\[ T(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]

- **client**
- **shard 1**
- **shard 2**
- **shard 3**

delete \(X_1, X_2\); create \(Y_1, Y_2\)
S-BAC

\[ T(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]
$T(x_1, x_2) \rightarrow (y_1, y_2, y_3)$
Atomix

\[ T(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]
Atomix

\[ T(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]
**Atomix**

\[
T(x_1, x_2) \rightarrow (y_1, y_2, y_3)
\]
Atomix

\[ T(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]
Atomix

$T(x_1, x_2) \rightarrow (y_1, y_2, y_3)$
Cross-Shard Consensus
How does it achieve linear scalability?
Insecure under parallel composition
Attacks

**Double spend any object**

- Does not need to collude with any node
- Acts as client or passive observer
- Re-orders network messages (not always needed)
Attack against S-BAC
Double-spend $X_1$

\[ T(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]
Attack against S-BAC
Double-spend $X_1$

$T'(\tilde{x}_1, x_2) \rightarrow (y_1, y_2, y_3)$
Attack against S-BAC
Double-spend $X_1$

$T'(\overline{x_1}, x_2) \rightarrow (y_1, y_2, y_3)$

$c \rightarrow \text{BFT} \rightarrow s1 \rightarrow \text{BFT} \rightarrow s2 \rightarrow \text{BFT} \rightarrow s3$
Attack against S-BAC

Double-spend $X_1$

$T'(\tilde{x}_1, x_2) \rightarrow (y_1, y_2, y_3)$

lock $X_2$
Attack against S-BAC

Double-spend $X_1$

$T'(x_1, x_2) \rightarrow (y_1, y_2, y_3)$

$BFT$  
\[ \text{pre-abort}(T') \]

$BFT$  
\[ \text{pre-accept}(T') \]

$c$

$s1$

$s2$

$s3$

lock $X_2$
Attack against S-BAC

Double-spend $X_1$

$T'(\tilde{x}_1, x_2) \rightarrow (y_1, y_2, y_3)$

$c$  
$s_1$  
$s_2$  
$s_3$  

lock $X_2$

$T(x_1, x_2) \rightarrow (y_1, y_2, y_3)$

$c$  
$s_1$  
$s_2$  
$s_3$  

$abort(T)$

$pre-accept(T')$

$pre-abort(T)$

$pre-accept(T)$

$pre-abort(T)$
Attack against S-BAC

Double-spend $X_1$

$T'(\tilde{x}_1, x_2) \rightarrow (y_1, y_2, y_3)$

$c$  \hspace{1cm} pre-abort(T')

$s_1$  \hspace{1cm} BFT

$s_2$  \hspace{1cm} BFT

$s_3$  \hspace{1cm} lock $X_2$

$\text{pre-accept}(T)$

from shard 1

$T(x_1, x_2) \rightarrow (y_1, y_2, y_3)$

$c$  \hspace{1cm} pre-accept(T)

$s_1$  \hspace{1cm} BFT

$s_2$  \hspace{1cm} BFT

$s_3$  \hspace{1cm} BFT

$\text{abort}(T)$

$\text{pre-abort}(T)$

Double-spend $X_1$
Attack against S-BAC

Double-spend $X_1$

\[ T'(\bar{x}_1, x_2) \rightarrow (y_1, y_2, y_3) \]

\[ T(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]

lock $X_2$

unlock $X_2$

abort($T'$)

pre-accept($T$) from shard 1

pre-accept($T$)

pre-abort($T$)

pre-accept($T$)
Attack against S-BAC
Double-spend $X_1$

$T^*(x_1) \rightarrow (y_*)$
Attack against S-BAC
Double-spend $X_1$

$T(x_1, x_2) \rightarrow (y_1, y_2, y_3)$

client

shard 1

shard 2

shard 3

attacker
Attack against S-BAC

Double-spend $X_1$

$T(x_1, x_2) \rightarrow (y_1, y_2, y_3)$
Attack against S-BAC

Double-spend $X_1$

$T(x_1, x_2) \rightarrow (y_1, y_2, y_3)$

client

shard 1

shard 2

shard 3

attacker

pre-abort($T$)

pre-accept($T$)

pre-accept($T$)

pre-accept($T$)
Attack against S-BAC
Double-spend $X_1$

Before attack

<table>
<thead>
<tr>
<th>X_1</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>X_2</td>
<td>5</td>
</tr>
</tbody>
</table>

After attack

<table>
<thead>
<tr>
<th>$Y_1$</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Y_2$</td>
<td>4</td>
</tr>
<tr>
<td>$Y_3$</td>
<td>10</td>
</tr>
</tbody>
</table>
If it is not implemented, it does not work
Attacks against S-BAC

First phase

client

shard 1

shard 2

shard 3
### Attacks against S-BAC

#### First phase

<table>
<thead>
<tr>
<th>Phase 1 of S-BAC</th>
<th>Shard 1 (potential victim)</th>
<th>Shard 2 (potential victim)</th>
<th>Phase 2 of S-BAC</th>
<th>Shard 1 (potential victim)</th>
<th>Shard 2 (potential victim)</th>
<th>Shard 3 (potential victim)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>pre-accept(T) lock x₁</td>
<td>pre-accept(T) lock x₂</td>
<td>accept(T)</td>
<td>accept(T)</td>
<td>create y₁; inactivate x₁</td>
<td>create y₂; inactivate x₂</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>accept(T)</td>
<td>accept(T)</td>
<td></td>
<td>create y₃</td>
</tr>
<tr>
<td>2</td>
<td>pre-abort(T)</td>
<td></td>
<td>accept(T)</td>
<td>create y₁; inactivate x₁</td>
<td>abort(T)</td>
<td>unlock x₂</td>
</tr>
<tr>
<td>3</td>
<td>pre-abort(T)</td>
<td></td>
<td>accept(T)</td>
<td>create y₂; inactivate x₂</td>
<td></td>
<td>create y₃</td>
</tr>
<tr>
<td>4</td>
<td>pre-abort(T)</td>
<td>pre-abort(T)</td>
<td>abort(T)</td>
<td>abort(T)</td>
<td>unlock x₂</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>pre-abort(T)</td>
<td>pre-accept(T)</td>
<td>abort(T)</td>
<td>abort(T)</td>
<td>unlock x₂</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>pre-accept(T)</td>
<td></td>
<td>abort(T)</td>
<td>accept(T)</td>
<td></td>
<td>create y₃</td>
</tr>
<tr>
<td>7</td>
<td>pre-accept(T)</td>
<td>pre-abort(T)</td>
<td>abort(T)</td>
<td>create y₁; inactivate x₁</td>
<td>abort(T)</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>pre-abort(T)</td>
<td></td>
<td>accept(T)</td>
<td>create y₂; inactivate x₂</td>
<td>abort(T)</td>
<td>create y₃</td>
</tr>
<tr>
<td>9</td>
<td>pre-abort(T)</td>
<td>pre-abort(T)</td>
<td>abort(T)</td>
<td>-</td>
<td>abort(T)</td>
<td>-</td>
</tr>
</tbody>
</table>

*Note: The highlighted rows indicate incorrect executions due to replay attacks.*
Attacks against S-BAC

Second phase
Attacks against S-BAC
Second phase

<table>
<thead>
<tr>
<th>Shard 1</th>
<th>Shard 2</th>
<th>Shard 3 (potential victim)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>accept($T$)</td>
<td>accept($T$)</td>
</tr>
<tr>
<td></td>
<td>create $y_1$; inactivate $x_1$</td>
<td>create $y_2$; inactivate $x_2$</td>
</tr>
<tr>
<td>2</td>
<td>$\triangleright$accept($T$)</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>$\triangleright$accept($T$)</td>
</tr>
<tr>
<td>4</td>
<td>$\triangleright$accept($T$)</td>
<td>$\triangleright$accept($T$)</td>
</tr>
<tr>
<td>5</td>
<td>abort($T$)</td>
<td>abort($T$)</td>
</tr>
<tr>
<td></td>
<td>(unlock $x_1$)</td>
<td>(unlock $x_2$)</td>
</tr>
<tr>
<td>6</td>
<td>$\triangleright$accept($T$)</td>
<td>$\triangleright$accept($T$)</td>
</tr>
<tr>
<td>7</td>
<td>$\triangleright$accept($T$)</td>
<td>$\triangleright$accept($T$)</td>
</tr>
<tr>
<td>8</td>
<td>$\triangleright$accept($T$)</td>
<td>$\triangleright$accept($T$)</td>
</tr>
</tbody>
</table>
What causes these issues?

**Issue 1.** Input shards cannot associate protocol messages to a specific protocol execution.

**Issue 2.** Output shards (that are not also input shards) do not experience the first phase of the protocol.
Easy Fix?

Global sequence numbers?  |  Wait for messages to arrive?

Linear Scalability  |  High Throughput
Byzcuit
Fix issue 1

Add sequence numbers per object
Byzcuit
Fix issue 2

 Dummy objects for output shards

\[ X_1, S_{x1} \]
Shard 1

\[ X_2, S_{x2} \]
Shard 2

\[ D3, S_{D3} \]
Shard 3
Byzcuit

\[ \{ S_T, T(x_1, x_2, d_3) \rightarrow (y_1, y_2, y_3) \} \]
Byzcuit

\[ \{ S_T, T(x_1, x_2, d_3) \rightarrow (y_1, y_2, y_3) \} \]

**Check 1.** Are all inputs active / transaction well formed?

**Check 2.** Is the sequence number \( S_T \)
\[ S_T \geq \max\{ S_{X1}, S_{X2} \} \]?
Byzcuit

\[ \{ S_T, T(x_1, x_2, d_3) \rightarrow (y_1, y_2, y_3) \} \]

client

if checks fail: \( S_{X1} \leftarrow S_T + 1 \)

shard 1

if checks fail: \( S_{X2} \leftarrow S_T + 1 \)

shard 2

if checks fail: \( S_{D3} \leftarrow S_T + 1 \)

shard 3

TM
Byzcuit

\[ \{ S_T, T(x_1, x_2, d_3) \rightarrow (y_1, y_2, y_3) \} \]

- **Client**
  - Otherwise: lock \( X_1 \), store \( (S_T, T) \)

- **Shard 1**
  - BFT
  - Otherwise: lock \( X_2 \), store \( (S_T, T) \)

- **Shard 2**
  - BFT
  - Otherwise: lock \( D_3 \), store \( (S_T, T) \)

- **Shard 3**
  - BFT

- **TM**
Byzcuit

\[ \{S_T, T(x_1, x_2, d_3) \rightarrow (y_1, y_2, y_3)\} \]

client

shard 1

shard 2

shard 3

TM

pre-accept(T, S_T)
\{ S_T, T(x_1, x_2, d_3) \rightarrow (y_1, y_2, y_3) \}
Byzcuit

\[ \{ S_T, T(x_1, x_2, d_3) \rightarrow (y_1, y_2, y_3) \} \]

client

shard 1

shard 2

shard 3

TM

if \((T, ST)\), inactivate \(X_1, X_2, D_3\) create \(Y_1, Y_2, Y_3\)
Why is Byzcuit secure?

**Issue 1.** Input shards cannot associate protocol messages to a specific protocol execution.

Sequence numbers:

act as session ID

**Issue 2.** Output shards (that are not also input shards) do not experience the first phase of the protocol.

Dummy objects:

all shards experience the first phase of the protocol
Anyone can be a TM
Byzcuit
Implementation

• Fork of Java Chainspace
• Based on BFT-SMART
• Only a prototype to demonstrate its properties

https://github.com/sheharbano/byzcuit
Figure 7: The effect of the number of shards on transaction throughput. Each transaction has 2 input objects and 5 output objects, both chosen randomly from shards.

Figure 8: Decrease of Byzcuit throughput with the number of dummy objects. Each transaction has 1 input object, and up to 5 dummy objects randomly selected from unique non-input shards. 6 shards are used.

Figure 9: Client-perceived latency vs. system load (number of transactions received per second by Byzcuit), for 6 shards with 2 inputs and 5 outputs per transaction (both chosen randomly from shards).

8 Conclusion

We presented the first replay attacks against cross-shard consensus protocols in sharded distributed ledgers. These attacks affect both shard-driven and client-driven consensus protocols, and allow attackers to double-spend or lock objects with minimal efforts. The attacker can act independently without colluding with any nodes, and succeed even if all nodes are honest; most of the attacks work also under asynchrony. While addressing these attacks seems like an implementation detail, their many variants illustrate that a fundamental re-think of cross-shard commit protocols is required to protect against them.

We developed Byzcuit, a new cross-shard consensus protocol merging features from shard-led and client-led consensus protocols, and withstanding replay attacks. Byzcuit can be seen as unifying Atomix (from Omniledger) and S-BAC (from Chainspace), into an O(n) protocol, that is efficient and secure. We implemented a prototype of Byzcuit and evaluated it on a real cloud-based testbed, showing that it is more performant than Chainspace, and on par with Omniledger performance. The resulting protocol is a drop-in replacement for either, and can be adopted to immunize systems based on those designs.
Byzcuit
Finality

Figure 7: The effect of the number of shards on transaction throughput. Each transaction has 2 input objects and 5 output objects, both chosen randomly from shards.

Transaction has 1 input object, and we vary the number of dummy objects from 1–5 selected from unique shards, resulting in a corresponding decrease in concurrency because as many shards end up processing the transaction. For example, 2 dummy objects means that 3 shards process the transaction (1 input shard, and 2 more shards corresponding to the dummy objects). As expected, the throughput decreases by 20–250 tps with the addition of each dummy object, and reaches 750 tps when all 6 shards handle all transactions.

Client-perceived Latency.

Figure 9 shows the client-perceived latency—the time from when a client submits a transaction, until it receives a decision from Byzcuit about whether the transaction has been committed—under varying system loads (expressed as transactions submitted to Byzcuit per second). We submit a total of 1200 transactions at 200–1000 transactions per second to Byzcuit with 6 shards. Each transaction has 2 input objects and 5 output objects, both chosen randomly from shards. When the system is experiencing a load of up to 1000 tps, clients hear back about their transactions in less than a second on average, even with our replay attack defenses.

8 Conclusion

We presented the first replay attacks against cross-shard consensus protocols in sharded distributed ledgers. These attacks affect both shard-driven and client-driven consensus protocols, and allow attackers to double-spend or lock objects with minimal efforts. The attacker can act independently without colluding with any nodes, and succeed even if all nodes are honest; most of the attacks work also under asynchrony. While addressing these attacks seems like an implementation detail, their many variants illustrate that a fundamental re-think of cross-shard commit protocols is required to protect against them.

We developed Byzcuit, a new cross-shard consensus protocol merging features from shard-led and client-led consensus protocols, and withstanding replay attacks. Byzcuit can be seen as unifying Atomix (from Omniledger) and S-BAC (from Chainspace), into an $O(n)$ protocol, that is efficient and secure. We implemented a prototype of Byzcuit and evaluated it on a real cloud-based testbed, showing that it is more performant than Chainspace, and on par with Omniledger performance. The resulting protocol is a drop-in replacement for either, and can be adopted to immunize systems based on those designs.
Open Questions

- Easy DoS by exhausting the sequence numbers
- Load balancing of objects
- The Mega transaction
Conclusion
Part I - Increasing Throughput

Byzcuit

- S-BAC + Atomix
- High throughput, linear scalability, BFT resilience, Fast finality

- **Paper:** https://arxiv.org/abs/1901.11218
- **Code:** https://github.com/sheharbano/byzcuit