Replay Attacks and Defenses Against Cross-shard Consensus in Sharded Distributed Ledgers

Authors
Alberto Sonnino*
Shehar Bano*
Mustafa Al-Bassam*
George Danezis*

* University College London

January 2019
The Team

Alberto Sonnino

Bano Shehar

Mustafa Al-Bassam

George Danezis
Blockchains' Scalability
Blockchains' Scalability

- Several ways to enable blockchain scalability

- **Sharding**
  - Diagram showing sharding concept

- **State Channels**
  - Diagram showing state channels concept

- **Bigger Blocks**
  - Diagram showing bigger blocks concept
Sharded Distributed Ledgers

- Linear scalability through sharding
Sharded Distributed Ledgers

- Linear scalability through state sharding

Diagram showing three shards labeled shard 1, shard 2, and shard 3.
Sharded Distributed Ledgers

- Linear scalability through state sharding
Sharded Distributed Ledgers

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

![Diagram showing X1, X2, and X3 shards with different states and connections between them. The diagram illustrates the concept of distributed ledgers with sharding, showing how transactions are processed across multiple shards.](image-url)
Sharded Distributed Ledgers

transaction

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

- What can the attacks do?
  - Double-spend any resource (e.g., coins); sometimes they can lock user's resources

- Threat Model: the attacker
  - does not need to collude with any node
  - acts as client or passive observer
  - re-orders network messages (only needed for some of the attacks)
Attacks Overview

- Easy to fix if

  Synchrony assumption for safety

  or

  Shards store & check old data (break scalability)
Attacks Overview

Illustration of the attacks

---

Chainspace

Chainspace: A Sharded Smart Contracts Platform

---

OmniLedger

Omniledger: A Secure, Scale-Out, Decentralized Ledger via Sharding

---

NDSS'18

S&P'18
Shard-Led Cross-Shard Consensus

- Chainspace

\[ T(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]
Shard-Led Cross-Shard Consensus

- **Chainspace**

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

Client \[\text{pre-accept}(T)\]

shard 1

BFT

shard 2

BFT

shard 3

\[\text{lock } X_1, X_2\]
Shard-Led Cross-Shard Consensus

- **Chainspace**

![Diagram](image.png)

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

- Client
- Shard 1
- Shard 2
- Shard 3

BFT

pre-accept(T)

delete $X_1, X_2$; create $Y_1, Y_2$
Shard-Led Cross-Shard Consensus

- **Chainspace**

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

**Figure 1:** Shard-Led Cross-Shard Consensus Protocol

**Figure 2:** Diagram of the cross-shard consensus protocol.
Shard-Led Cross-Shard Consensus

- **Chainspace**

![Diagram](image-url)

The diagram illustrates the process of a transaction `T(x_1, x_2) → (y_1, y_2, y_3)` involving a client and three shards. The transaction is processed in two phases:

**First Phase**
- The client submits the transaction to shard 1.
- Shard 1 processes the transaction and sends a pre-accept message to shards 2 and 3.

**Second Phase**
- Shards 2 and 3 also process the transaction and send pre-accept messages to the other shards.
- The shards eventually reach a consensus and send accept messages back to the client.

Note: The diagram shows the interaction between the client and the shards, highlighting the consensus process in a cross-shard environment.
Shard-Led Cross-Shard Consensus

- First phase attacks

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

- First phase attacks: let's double-spend $X_1$

transaction

$$T(x_1, x_2) \rightarrow (y_1, y_2, y_3)$$
Shard-Led Cross-Shard Consensus

- First phase attacks: let's double-spend $X_1$

pre-accept(T)  

from shard 1
Shard-Led Cross-Shard Consensus

- First phase attacks: let's double-spend $X_1$

(bad) transaction

$$T'(\widetilde{x_1}, x_2) \rightarrow (y_1, y_2, y_3)$$
Shard-Led Cross-Shard Consensus

- First phase attacks: recording messages

\[ T'(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]
Shard-Led Cross-Shard Consensus

- First phase attacks: recording messages

\[ T'(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]
Shard-Led Cross-Shard Consensus

- First phase attacks: recording messages

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

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

(lock \( X_2 \))
Shard-Led Cross-Shard Consensus

- First phase attacks: recording messages

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

\[ \text{lock } X_2 \]

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

\[ \text{pre-accept}(T) \]

\[ \text{from shard 1} \]

\[ \text{abort}(T) \]

(because \( X_2 \) is locked)
Shard-Led Cross-Shard Consensus

- First phase attacks: recording messages

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

\[ \text{pre-abort}(T) \]

\[ \text{pre-accept}(T) \]

\[ \text{lock } X_2 \]

\[ \text{unlock } X_2 \]

\[ \text{abort}(T) \]

\[ \text{from shard 1} \]

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

\[ \text{pre-accept}(T) \]

\[ \text{pre-abort}(T) \]

\( \text{because } X_2 \text{ is locked} \)
Shard-Led Cross-Shard Consensus

- First phase attacks: spend $X_1$

\[ T^*(x_1) \rightarrow (y_*) \]

client

shard 1

BFT

10

pre-accept(T) from shard 1
Shard-Led Cross-Shard Consensus

- **First phase attacks: double-spend** \( x_1 \)

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

- Client
- Shard 1
- Shard 2

![Diagram](image-url)
Shard-Led Cross-Shard Consensus

- First phase attacks: 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-accept(T) from shard 1

pre-accept(T)

pre-abort(T)

accept(T)

abort(T)

BFT

BFT

BFT

BFT

BFT
4.3 Attacks on the First Phase of S-BAC

- First phase attacks: double-spend $X_1$

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

Client 

shard 1 

shard 2 

shard 3 

Attacker
Shard-Led Cross-Shard Consensus

**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>(\triangleright accept(T))</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>-</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>(\triangleright accept(T))</td>
</tr>
<tr>
<td>8</td>
<td>(\triangleright accept(T))</td>
<td>(\triangleright accept(T))</td>
</tr>
</tbody>
</table>
Shard-Led Cross-Shard Consensus

- Second phase

\[ T(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]
Client-Led Cross-Shard Consensus

- Omniledger

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

client

\( \text{shard 1} \)

\( \text{shard 2} \)

\( \text{shard 3} \)

pre-accept(T)

inactivate \( X_1, X_2 \)
Client-Led Cross-Shard Consensus

○ Omniledger

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

create \( Y_1, Y_2, Y_3 \)
Client-Led Cross-Shard Consensus

Omniledger

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

first phase

second phase
Client-Led Cross-Shard Consensus

**First phase attacks**

<table>
<thead>
<tr>
<th>Shard 1 (potential victim)</th>
<th>Shard 2 (potential victim)</th>
<th>Client (victim)</th>
<th>Shard 1 (potential victim)</th>
<th>Shard 2 (potential victim)</th>
<th>Shard 3 (potential victim)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre-accept(T)</td>
<td>pre-accept(T)</td>
<td>accept(T)</td>
<td>-</td>
<td>create y₁</td>
<td>create y₂</td>
</tr>
<tr>
<td>pre-abort(T)</td>
<td>abort(T)</td>
<td>-</td>
<td>re-activate x₁</td>
<td>re-activate x₂</td>
<td>-</td>
</tr>
<tr>
<td>pre-abort(T)</td>
<td>abort(T)</td>
<td>-</td>
<td>re-activate x₁</td>
<td>re-activate x₂</td>
<td>-</td>
</tr>
<tr>
<td>pre-abort(T)</td>
<td>pre-abort(T)</td>
<td>abort(T)</td>
<td>-</td>
<td>re-activate x₁</td>
<td>re-activate x₂</td>
</tr>
<tr>
<td>pre-abort(T)</td>
<td>pre-accept(T)</td>
<td>abort(T)</td>
<td>-</td>
<td>-</td>
<td>re-activate x₂</td>
</tr>
<tr>
<td>pre-accept(T)</td>
<td>accept(T)</td>
<td>-</td>
<td>create y₁</td>
<td>create y₂</td>
<td>create y₃</td>
</tr>
<tr>
<td>pre-accept(T)</td>
<td>pre-abort(T)</td>
<td>abort(T)</td>
<td>-</td>
<td>re-activate x₁</td>
<td>-</td>
</tr>
<tr>
<td>pre-abort(T)</td>
<td>pre-abort(T)</td>
<td>abort(T)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>pre-accept(T)</td>
<td>accept(T)</td>
<td>-</td>
<td>create y₁</td>
<td>create y₂</td>
<td>create y₃</td>
</tr>
</tbody>
</table>

Table 3: Cross-Shard replay attacks.

Table 4: Cross-Shard attacks on the second phase of Atomix.
Client-Led Cross-Shard Consensus

- First phase attacks: let's double-spend $x_1$

\[ T(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]
Client-Led Cross-Shard Consensus

- First phase attacks: let's double-spend $X_1$

pre-accept(T)

from shard 1
Client-Led Cross-Shard Consensus

- First phase attacks: let's double-spend $X_1$

(bad) transaction

$T'(\widetilde{x}_1, x_2) \rightarrow (y_1, y_2, y_3)$
Client-Led Cross-Shard Consensus

- First phase attacks: recording messages

\[ T'(\overline{x}_1, x_2) \rightarrow (y_1, y_2, y_3) \]
Client-Led Cross-Shard Consensus

- First phase attacks: recording messages

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

Invalidate \( X_2 \)
Client-Led Cross-Shard Consensus

- First phase attacks: recording messages

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

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

invalidate \( X_2 \)

(because \( X_2 \) is invalidated)
Client-Led Cross-Shard Consensus

- First phase attacks: recording messages

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

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

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

\( \text{client} \)

\( \text{shard 1} \)

\( \text{BFT} \)

\( \text{shard 2} \)

\( \text{BFT} \)

\( \text{invalidate } X_2 \)

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

\( \text{from shard 1} \)

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

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

\( \text{abort}(T) \)

\( \text{client} \)

\( \text{shard 1} \)

\( \text{BFT} \)

\( \text{shard 2} \)

\( \text{BFT} \)

\( \text{shard 2} \)

\( \text{BFT} \)

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

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

\( \text{client} \)

\( \text{shard 1} \)

\( \text{shard 2} \)

\( \text{BFT} \)

\( \text{BFT} \)

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

\( \text{abort}(T) \)

\( \text{(because } X_2 \text{ is invalidated)} \)
Client-Led Cross-Shard Consensus

- First phase attacks: recording messages

\[ T'(\widetilde{x_1}, x_2) \rightarrow (y_1, y_2, y_3) \]

\[ \text{pre-abort}(T) \]

\[ \text{pre-accept}(T) \]

\[ \text{client} \rightarrow \text{shard 1} \rightarrow \text{shard 2} \]

\[ \text{invalidate } X_2 \]

\[ \text{abort}(T) \]

\[ \text{pre-accept}(T) \]

\[ \text{from shard 1} \]

\[ \text{re-create } X_2 \]

\[ \text{because } X_2 \text{ is invalidated} \]
Client-Led Cross-Shard Consensus

- First phase attacks: spend $X_1$

$$T^*(x_1) \rightarrow (y_*)$$

Client

Shard 1

BFT

Pre-accept(T) from shard 1
Client-Led Cross-Shard Consensus

- First phase attacks: double-spend $X_1$

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

- client
- shard 1
  - BFT
  - pre-abort(T)
- shard 2
  - BFT
  - pre-accept(T)
- shard 3
- attacker

pre-accept(T) from shard 1
Client-Led Cross-Shard Consensus

- First phase attacks: double-spend $X_1$

\[ T(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]
Client-Led Cross-Shard Consensus

- **Second phase attacks**

<table>
<thead>
<tr>
<th>Client</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>accept($T$)</td>
<td>-</td>
<td>create $y_3$</td>
</tr>
<tr>
<td></td>
<td>create $y_1$</td>
<td>create $y_2$</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>▷ abort($T$)</td>
<td>re-activate $x_1$</td>
<td>re-activate $x_2$</td>
</tr>
<tr>
<td>3</td>
<td>abort($T$)</td>
<td>re-activate $x_1$</td>
<td>re-activate $x_2$</td>
</tr>
<tr>
<td>4</td>
<td>▷ accept($T$)</td>
<td>create $y_1$</td>
<td>create $y_2$</td>
</tr>
</tbody>
</table>
Client-Led Cross-Shard Consensus

- Second phase attacks

\[ T(x_1, x_2) \rightarrow (y_1, y_2, y_3) \]
Fixing replay attacks without breaking scalability

- What issues lead to those replay attacks?

  **Issue 1.** Input shards cannot associate protocol messages to a specific instance of a transaction.
Fixing replay attacks without breaking scalability

- What issues lead to those replay attacks?

**Issue 1.** Input shards cannot associate protocol messages to a specific instance of a transaction.

**Issue 2.** Output shards (that are not also input shards) do not experience the first phase of the protocol.
Fixing replay attacks without breaking scalability

Byzcuit

Chainspace  ❤  Omniledger
Byzcuit

- Fixing issue 1: adding sequence numbers per object

\[ X_1, S_{x_1} \]

\[ X_2, S_{x_2} \]
Byzcuit

- Fixing issue 2: dummy objects for output shards
Byzcuit

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

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

- client
- shard 1
- shard 2
- shard 3

Figure 5: An example execution of Byzcuit for a valid transaction

Figure 6: State machine representing the life cycle of objects in Byzcuit.
Byzcuit

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

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

Check 2. Is the sequence number \( S_T \)

\[ S_T \geq max\{S_{X1}, S_{X2}\} \]?
Client makes a transaction to shard 1, 2, or 3

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

If Check fail:
Update all sequence numbers

\[ S_{X1} \leftarrow S_T + 1 \]
\[ S_{X2} \leftarrow S_T + 1 \]
\[ S_{D3} \leftarrow S_T + 1 \]
Byzcuit

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

If Check pass:
1. Lock objects as Chainspace
2. Store the session ID \((S_T, T)\)

lock \(X_1, X_2, D_3\)
Byzcuit

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

client

shard 1

shard 2

shard 3

TM

pre-accept(\( T, s_T \))
Byzcuit

An example execution of Byzcuit for a valid transaction

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

client

shard 1

BFT

shard 2

BFT

shard 3

BFT

TM

pre-accept(\(T, s_T\))

accept(\(T, s_T\))

If \((S_T, T)\)
date X_1, X_2, D_3
create Y_1, Y_2, Y_3, D_3'}
Byzcuit

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

client

shard 1

BFT

shard 2

BFT

shard 3

BFT

TM

\[ \text{pre-accept}(T, s_T) \]

\[ \text{accept}(T, s_T) \]

first phase

second phase
Byzcuit

- The transaction manager (TM)

Anyone can be a TM: it does not operate on the basis of any secret, and has no discretion in the protocol.

- The TM can be a shard
  Input shards contact in turn each node of the TM shard until they find a honest node

- The TM can be a single entity
  If the TM dies, anyone can take over: liveness is guaranteed as long as there is one honest party in the system
Byzcuit

• How does it prevents replay attacks

**Issue 1.** Input shards cannot associate protocol messages to a specific instance of a transaction.
Byzcuit

• How does it prevents replay attacks

Issue 1. Input shards cannot associate protocol messages to a specific instance of a transaction.

Sequence numbers: they act as session ID
Byzcuit

- How does it prevents replay attacks

**Issue 1.** Input shards cannot associate protocol messages to a specific instance of a transaction.

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

Sequence numbers: they act as session ID
Byzcuit

- How does it prevents replay attacks

**Issue 1.** Input shards cannot associate protocol messages to a specific instance of a transaction.

| Sequence numbers: they 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 |
Byzcuit

- Performance

Open Source

https://github.com/sheharbano/byzcuit
Byzcuit

- Performance

**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 throughput decreases by 20–25% with the addition of each dummy object, reaching 750 tps when all 6 shards handle all transactions.

**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).

**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 succeeds even if all nodes are honest; most of the attacks work under asynchrony. While addressing these attacks seems like an implementation detail, their many variants illustrate that a fundamental rethink 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

- Performance

![Graph showing the effect of the number of shards on transaction throughput. Each transaction has 1 input object and up to 5 dummy objects randomly selected from unique non-input shards. 6 shards are used.](image)

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.

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 inputs 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.
Byzcuit

Performance

(2 input ; 5 outputs ; 6 shards)
Conclusion

- Replay attacks against sharded distributed ledgers
- Fix without additional synchrony assumption / breaking scalability
- Importance of implementation and evaluation
Thank you for your attention

Questions?