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

Alberto Sonnino[1,2], Shehar Bano[1,2], Mustafa Al-Bassam[1,2] and George Danezis[1,2]

[1]University College London
[2]`chainspace.io`

Abstract

We present the first replay attacks against sharded distributed ledgers. These attacks target cross-shard consensus protocols allowing an attacker to double-spend or lock resources 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 also work under asynchrony. These attacks are effective against both shard-led and client-led cross-shard consensus approaches. We presented Byzcuit—a new cross-shard consensus protocol that withstands those attacks.

1 Introduction

Sharding is a promising solution to blockchain scalability issues, and a growing number of systems are implementing sharded blockchains [2]. The key idea is to create groups (or shards) of nodes that handle only a subset of all the transactions, relying on classical Byzantine Fault Tolerance (BFT) protocols for reaching intra-shard consensus. These systems achieve optimal performance and scalability because: (*i*) non-conflicting transactions can be processed in parallel by multiple shards; and (*ii*) the system can scale up *via* creation of new shards. However, this separation of transaction handling across shards is not perfectly 'clean'—a transaction might rely on data managed by multiple shards, requiring an additional step of cross-shard consensus across the concerned shards. Typically an atomic commit protocol (such as the two-phase commit protocol [5]) is run across all the concerned shards to ensure that the transaction is accepted by all or none of those shards.

In this paper, we present the first replay attacks on cross-shard consensus in sharded blockchains. An attacker can launch these attacks with minimal effort, without subverting any nodes, and assuming a weakly synchronous (and in some cases, asynchronous) network—even when the byzantine safety assumptions are satisfied. These attacks compromise key system properties of safety and liveness, effectively enabling the attacker to double-spend coins (or any other objects managed by the blockchain) and create coins out of thin air. Our attacks apply to the two main approaches to achieve cross-shard consensus [2]: (*i*) shard-led protocols that only involve the concerned shards, and require no external entity for coordination (Section 3.1); and client-led protocols that are coordinated by the client (Section 4.1). We concretely sketch the replay attacks in the context of two representative systems: Chainspace [1] (Network and Distributed Systems Security Symposium 2018) as an example of shard-led protocols; and Omniledger [6] (IEEE Symposium on Security and Privacy 2018) as an example of client-led protocols. However, the attacks are generic and apply to other systems that are based on similar models. We also provide a comparison with mutex-based cross-shard consensus protocols used by Ethereum in Appendix A. For each of the two cross-shard consensus approaches, we describe how an attacker can actively stage the attack by eliciting from the system the messages to replay (in contrast to passively observing the network traffic, and waiting to detect and record the target messages). We also discuss the feasibility of these attacks and their real-world impact.

Drawing insights from our analysis of performance tradeoffs and replay attack vulnerability in shard-led and client-led cross-shard consensus protocols, we present a hybrid system Byzcuit (Section 5) that combines useful features from both these design approaches. Byzcuit employs a Transaction Manager to coordinate cross-shard communication, reducing its cost to $O(n)$ in the happy case. We build additional design features into Byzcuit to make it resilient to the replay attacks presented in this paper.

In summary, this paper has two key contributions: (*i*) developing the first replay attacks against shard-led and client-led cross-shard consensus protocols; and (*ii*) designing a hybrid, new system Byzcuit with improved performance tradeoffs, and which is resilient against the replay attacks.

2 Background and Overview

We present background on cross-shard atomic commit protocols, and provide an overview of our replay attacks.

2.1 Cross-Shard Atomic Commit Protocols

We describe sharded blockchains and cross-shard consensus protocols.

Sharded blockchains. The blockchain is a decentralized, replicated, immutable and tamper-evident log—maintained by computers (called nodes) that form a distributed network. The blockchain only supports read and write operations; data on the blockchain cannot be deleted. Anyone can read data from the blockchain and verify its correctness. Only special node(s) can write to the blockchain by means of a consensus protocol, to ensure that the entire network agrees on new state of the blockchain as a result of the write operation.

Earlier systems like Bitcoin [7] probabilistically elect a single node which can extend the blockchain. However, such systems have low consistency (*i.e.*, forks can be created) and low performance (*i.e.*, high latency and low throughput). Consequently, there has been a shift to committee-based designs [2] where a group of nodes collectively extends the blockchain typically *via* classical byzantine fault tolerance (BFT) consensus protocols such as PBFT [4]. While these systems offer better performance, single-committee consensus is not scalable—as every node handles every transaction, adding more nodes to the committee decreases throughput due to the increased communication overhead.

This motivated the design of *sharded* systems, where multiple committees handle a subset of all the transactions—allowing parallel execution of transactions. Every committee has its own blockchain and set of objects (or unspent transaction outputs, UTXO) that they manage. Committees run an 'intra-shard' consensus protocol (*e.g.*, PBFT) within themselves, and extend their blockchains in parallel.

Cross-shard consensus. In sharded systems, some transactions may operate on objects handled by different shards, effectively requiring the relevant shards to additionally run a *cross-shard consensus protocol* to enable agreement across the shards. Specifically, if any of the shards relevant to the transaction rejects the transaction, all the other shards should likewise reject the transaction.

The typical choice for implementing cross-shard consensus is the two-phase atomic commit protocol [5]. This protocol has two phases which are run by a *coordinator*. In the first *voting* phase, the nodes tentatively write changes locally and report their status to the coordinator. If the coordinator does not receive status message from a node (*e.g.*, because the node crashed or the status message was lost), it assumes that the node's local write failed and sends a rollback message to all the nodes to ensure any local changes are reversed. If the coordinator receives status messages from all the nodes, it initiates the second *commit* phase and sends a commit message to all the nodes so they can permanently write the changes. In the context of sharded blockchains, the atomic commit protocol operates on shards (which make the local changes associated with the voting phase *via* an intra-shard consensus protocol like PBFT), rather than nodes. Another important consideration in the context of sharded blockchains is who will assume the role of the coordinator. There are currently two key approaches [2]: (*i*) the client acts as a coordinator; or (*ii*) the shards collectively assume the role of a coordinator.

2.2 Attack Overview

In Sections 3 and 4, we discuss replay attacks on both shard-led and client-led cross-shard consensus protocols, respectively. In this section, we provide a high-level description of these attacks and the threat model, and describe the notation we use.

Replay Attacks on Cross-Shard Consensus. The attacker records a target shard's responses to either the voting or the commit phase of the atomic commit protocol, and replays them during another instance of the atomic commit protocol. We present two families of replay attacks: (*i*) attacks against the first phase (*voting*) of the atomic commit protocol, and (*ii*) attacks against the second phase (*commit*) of the protocol.

To attack the first *voting* phase of the atomic commit protocol, the attacker replaces messages generated by the target shard by replaying prerecorded messages. In practice, the attacker does not replace those messages—it achieves a similar result by making its replayed messages arrive at the coordinator faster (racing the target shard's original message) exploiting the fact that the coordinator makes progress based on the first message it receives. Replaying messages in this fashion enables the attacker to compromise the system safety (by creating inconsistent state on the shards) and/or liveness (by causing valid transactions to be rejected).

To attack the second *commit* phase of the atomic commit protocol, the attacker simply replays prerecorded messages to target shards, and compromises consistency. The attacker can reply those messages at any time of its choice, and does not rely on any racing condition as in the previous case.

Threat Model. The attacker can successfully launch the previously described attacks independently (*i.e.*, without colluding with any shard nodes), and under the BFT honest majority safety assumption for nodes within shards (*i.e.*, the attacks are effective even if *all* nodes are honest). The replay attacks described in this paper are based on an attacker that can observe and record messages generated by shards. The attacker can be an external observer that passively collects the target messages, or it can act as a client and actively interact with the system to elicit the target messages. The attacks against the first phase of the atomic commit protocol

(Sections 3.3 and 4.3) assumes a weakly synchronous network which is exploited by the attacker to delay messages and race target shards by replaying prerecorded messages. The attacks against the second phase of the atomic commit protocol (Section 3.4 and 4.4) do not make any synchrony assumptions on the underlying network, and are effective even in an asynchronous network.

Notation. Operations on the blockchain are specified as *transactions*. A transaction defines some transformation on the blockchain state, and has input and output *objects*. An object is some data managed by the blockchain, such as a bank account or a hotel room. For example, $T(x_1, x_2) \rightarrow (y_1, y_2, y_3)$ represents a transaction with two inputs, x_1 managed by *shard* 1 and x_2 managed by *shard* 2; and three outputs, y_1 managed by *shard* 1, y_2 managed by *shard* 2, and y_3 managed by *shard* 3. We call the shards that manages the input objects *input shards* and the shards that manage the output objects *output shards*. It is possible for a shard to be both the input and output shard. Objects can be in two states: *active* (on unspent) objects are available for being processed by a transaction; and *inactive* (or spent) objects cannot be processed by any transaction. Additionally, some systems also associate "locked" state with objects that are currently being processed by a transaction to protect against manipulation by other concurrent transactions involving those objects. The attacks we describe in this paper generalise to transactions with k inputs and k' outputs managed by an arbitrary number of shards.

3 Shard-led Cross-shard Consensus

In shard-led cross-shard consensus protocols, the shards collectively take on the role of the coordinator in the atomic commit protocol. We describe replay attacks on shard-led cross-shard consensus protocols. To make the discussion concrete, we illustrate these attacks in the context of Chainspace [1] (Section 3.1), though we note that these attacks can be generalised to other similar systems. We discuss how the attacker can record shard messages to replay in future attacks (Section 3.2). In Sections 3.3 and 3.4, we describe replay attacks on the first and second phase of the cross-shard consensus protocol, followed by a discussion on the real-world impact of these attacks (Section 3.5).

3.1 Chainspace Overview

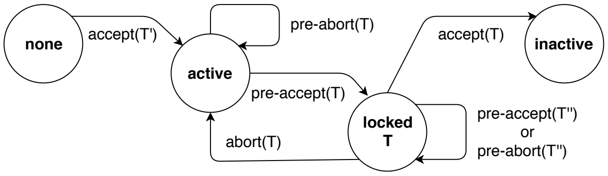

Figure 1: State machine representing the life cycle of Chainspace objects. An object becomes 'active' as a result of a previous successful transaction. The object state changes to 'locked' if a shard locally emits pre-accept(T) in the first phase of the cross-shard consensus protocol for a transaction T. A locked object cannot be processed by other transactions T''. If the second phase of the protocol results in accept(T), the object becomes 'inactive'; alternatively, if the result is abort(T) the object becomes 'active' again and is available for being processed by other transactions.

Chainspace uses a shard-led cross-shard consensus protocol called S-BAC. The client submits a transaction to the input shards. Each shard internally runs a BFT protocol to tentatively decide whether to accept or abort the transaction locally, and broadcasts its local decision (pre-accept(T) or pre-abort(T)) to other relevant shards. Figure 1 shows the state machine representing the life cycle of objects in Chainspace. A shard generates pre-abort(T) if the transaction fails local checks (*e.g.*, if any of the input objects are 'inactive' or 'locked'). If a shard generates pre-accept(T), it changes the state of the input objects to 'locked'. This is the first step of S-BAC, and is equivalent to the voting phase in the two-phase atomic commit protocol (Section 2.1).

Each shard collects responses from other relevant shards, and commits the transaction if all shards respond with pre-accept(T), or aborts the transaction otherwise. This is the second step of S-BAC, and is equivalent to the commit phase in the two-phase atomic commit protocol (Section 2.1). The shards communicate this decision to the client as well as the output shards by sending them the accept(T) or abort(T) messages. If the shard's decision is accept(T), it changes the input object state to 'inactive'. If the shard's decision is abort(T), it changes the input object state to 'active' (effectively unlocking it). Upon reception of the accept(T), the client concludes that the transaction was committed, and the output shard creates the output objects (with the state 'active') of the transaction.

Figure 2 shows an example execution of S-BAC for a valid transaction $T(x_1, x_2) \rightarrow (y_1, y_2, y_3)$ with two inputs (x_1 and x_2, both are active) and three outputs (y_1, y_2, y_3), where the final decision is accept(T). The client submits T to *shard* 1 and *shard* 2. Upon reception of T, both *shard* 1 and *shard* 2 confirm that the transaction is well-formed and the inputs objects are active, and emit pre-accept(T) at the end of the first phase of S-BAC. Each shard receives pre-accept(T) from the other shard, and emits accept(T) at the end of the second phase of S-BAC. As a result, the input objects x_1 and x_2 become inactive, and the output shards respectively create objects y_1, y_2, and y_3.

3.2 Message Recording

Prior to the replay attacks, the attacker records responses generated by shards. The attacker can record shard responses in the first phase of S-BAC (*i.e.*, pre-accept(T) or pre-abort(T)), enabling the family of attacks described in Section 3.3. The attacker can also record shard responses in the

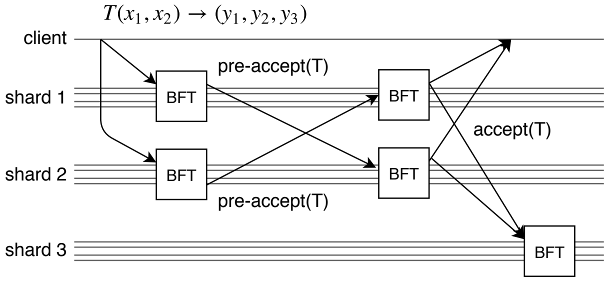

Figure 2: An example execution of S-BAC for a valid transaction $T(x_1,x_2) \rightarrow (y_1,y_2,y_3)$ with two inputs (x_1 and x_2, both are active) and three outputs (y_1,y_2,y_3), where the final decision is accept(T).

second phase of S-BAC (*i.e.*, accept(T) or abort(T)), enabling the family of attacks described in Section 3.4.

In the general case, the attacker passively collects the messages either by sniffing on protocol executions, or by downloading the blockchain and selecting the messages to replay.

Eliciting messages to replay. The attacker can also act as (or collude with) a client to actively elicit the target messages to record and later replay. This empowers the attacker to actively orchestrate the attacks. We describe how the attacker can trigger target messages in the context of an example, without loss of generality. Lets assume that *shard* 1 manages objects x_1 ('active') and object $\widetilde{x_1}$ ('inactive' or non-existent), and *shard* 2 manages object x_2 ('active'); $\widetilde{x*}$ means any inactive object on the shard, and $y*$ means any output object (*i.e.*, their details do not matter).

To elicit pre-accept(T) for a transaction $T(x_1,x_2) \rightarrow (y*)$ (the output $y*$ is not relevant here) from *shard* 1, the key consideration is to closely precede the transaction with another transaction T' that: (*i*) locks the inputs managed by at least one other shard (in this case x_2 on *shard* 2); and (*ii*) to ensure that the preceding transaction T' gets ultimately aborted, and x_2 becomes active again. The steps look as follows:

- The attacker submits $T'(x_2,\widetilde{x*}) \rightarrow (y*)$ to *shard* 2. This locks x_2.
- The attacker quickly follows up by submitting $T(x_1,x_2) \rightarrow (y*)$ to *shard* 1 and *shard* 2. *Shard* 1 generates pre-accept(T), which is the target message that the attacker records. *Shard* 2 generates pre-abort(T) because x_2 is locked by T'. Consequently, in the second phase of S-BAC, both *shard* 1 and *shard* 2 end up aborting T.
- T' is eventually aborted, making x_2 active again.

To elicit pre-abort(T) for a transaction $T(x_1,x_2) \rightarrow (y*)$ (the output $y*$ is not relevant here) from *shard* 1, the key consideration is to closely precede the transaction with another transaction T' that locks the input managed by the shard (in this case x_1 on *shard* 1). The steps look as follows:

- The attacker submits $T'(x_1,\widetilde{x*}) \rightarrow (y*)$ to *shard* 1. This locks x_1.
- The attacker quickly follows up by submitting $T(x_1,x_2) \rightarrow (y*)$ to *shard* 1 and *shard* 2. *Shard* 1 generates pre-abort(T) because x_1 is locked by T', which is the target message that the attacker records. *Shard* 2 generates pre-accept(T). Consequently, in the second phase of S-BAC, both *shard* 1 and *shard* 2 end up aborting T.
- T' is eventually aborted, making x_1 active again.

To elicit accept(T) used by the attacks described in Section 3.4, the attacker simply submits transaction T and observes and records its successful execution. The attacker has no incentive to record abort(T) messages as these are ignored by shards (see Table 2).

3.3 Attacks on the First Phase of S-BAC

We present replay attacks on the first phase of S-BAC by taking the example of a transaction $T(x_1,x_2) \rightarrow (y_1,y_2,y_3)$ as described in Section 2.2. These attacks easily generalise to transactions with k inputs and k' outputs managed by an arbitrary number of shards. The replay attacks work in two steps; (*i*) the attacker records pre-accept(T) or pre-abort(T) messages (as described in Section 3.2); and (*ii*) then replays those messages.

Table 1 shows the replay attacks that the attacker can launch, for all possible combinations of messages emitted by *shard* 1 and *shard* 2 in the first phase of S-BAC. The caption includes details about how to interpret this table. We describe row 6 of Table 1, to help readers interpret rest of the table on their own. In the correct execution (row 5), *shard* 1 and *shard* 2 emit pre-abort(T) (because x_1 is not active) and pre-accept(T) in the first phase, respectively. In the second phase, both shards emit abort(T) and the protocol terminates. Figure 3 illustrates the replay attack corresponding to row 6 of Table 1. The attacker races *shard* 1 by sending to *shard* 2 the prerecorded pre-accept(T) message from *shard* 1. As a result, *shard* 2 emits accept(T), inactivates object x_2 and creates object y_2. This leads to inconsistent state across the shards. In a correct execution: (*i*) if T is accepted all its inputs (x_1 and x_2) should become inactive, and all the outputs (y_1, y_2, y_3) should be created; and (*ii*) if T is aborted, all its inputs (x_1 and x_2) should become active again, and none of the outputs (y_1, y_2, y_3) should be created. However, here we have an incorrect termination of S-BAC: at the end of the protocol x_1 is active and x_2 is inactive; y_1 is not created, y_2 and y_3 are created.

Table 1 shows that through careful selection of the messages to replay, the attacks can be effective against any shard. All the attacks (except row 4) compromise consistency; the attacker can trick the input shards to inactivate arbitrary objects, and trick the output shards into creating new objects in

	Phase 1 of S-BAC		Phase 2 of S-BAC		
	Shard 1 (potential victim)	**Shard 2** (potential victim)	**Shard 1** (potential victim)	**Shard 2** (potential victim)	**Shard 3** (potential victim)
1	pre-accept(T) lock x_1	pre-accept(T) lock x_2	accept(T) create y_1; inactivate x_1	accept(T) create y_2; inactivate x_2	- create y_3
2	▷pre-abort(T)		accept(T) create y_1; inactivate x_1	abort(T) unlock x_2	- create y_3
3		▷pre-abort(T)	abort(T) unlock x_1	accept(T) create y_2; inactivate x_2	- create y_3
4	▷pre-abort(T)	▷pre-abort(T)	abort(T) unlock x_1	abort(T) unlock x_2	-
5	pre-abort(T) -	pre-accept(T) lock x_2	abort(T) -	abort(T) unlock x_2	-
6	▷pre-accept(T)		abort(T) -	accept(T) create y_2; inactivate x_2	- create y_3
7	pre-accept(T) lock x_1	pre-abort(T) -	abort(T) unlock x_1	abort(T) -	-
8		▷ pre-accept(T)	accept(T) create y_1; inactivate x_1	abort(T) -	- create y_3
9	pre-abort(T) -	pre-abort(T) -	abort(T) -	abort(T) -	-

Table 1: List of replay attacks against the first phase of S-BAC for all possible executions of the transaction $T(x_1,x_2) \rightarrow (y_1,y_2,y_3)$ as described in Section 2.2. The highlighted rows indicate correct executions of S-BAC (*i.e.*, without the attacker), and the other rows indicate incorrect executions due to the replay attacks. In multirows, the top sub-rows show the protocol messages emitted by shards, and the bottom sub-rows indicate local shard actions as a result of emitting those messages. For example, (column 3, row 2) means that *shard* 1 emits accept(T) (top sub-row), and creates a new object y_1 and inactivates x_1 (bottom sub-row). The first two columns indicate the messages emitted by each shard at the end the first phase of S-BAC. The attacker races shards at the end of the first phase of S-BAC by replaying prerecorded messages, marked with the symbol ▷ in the first two columns of Table 1. For example ▷pre-abort(T) at (column 1, row 2) means that the attacker sends to other relevant shards (in this case *shard* 2) a prerecorded pre-abort(T) message impersonating *shard* 1 that races the original pre-accept(T) (column 1, row 1) emitted by *shard* 1. The last three columns indicate the messages emitted at the end of the second phase of S-BAC.

violation of the protocol. The attack depicted in row 4 only affects availability.

3.4 Attacks on the Second Phase of S-BAC

We present replay attacks on the second phase of S-BAC. The attacker prerecords accept(T) messages as described in Section 3.2.

Table 2 shows replay attacks for all possible combinations of messages emitted by *shard* 1 and *shard* 2 in the second phase. Since the attacks we describe in this section assume that the first phase of S-BAC concluded correctly (*i.e.*, all the relevant shards unanimously decide to accept or reject a transaction), both the shards generate abort(T) (row 1) or accept(T) (row 5). The caption includes details about how to interpret this table. We describe row 6 of Table 2, to help readers interpret rest of the table on their own. In the correct execution (row 5), both the shards emit abort(T) and no output objects are created. In the attack in row 6, the attacker replays a prerecorded accept(T) from *shard* 1 to all the relevant shards (in this case *shard* 3). Upon receiving this message, *shard* 3 (incorrectly) creates y_3.

The potential victims of replay attacks corresponding to the second phase of S-BAC are the shards that *only* act as output shards (*i.e.*, do not simultaneously act as input shards). The attacker can replay accept(T) multiple times tricking *shard* 3 into creating y_3 multiple times. These attacks are possible because shards do not keep records of inactive objects (following the UTXO model) for scalability reasons, and because *shard* 3 takes part in only the second phase of S-BAC. The attacker can double-spend y_3 repeatedly by replaying a single prerecorded message multiple times, and spending the object (and effectively purging it from *shard* 3's UTXO) before each replay.

Contrarily to the attacks against the first phase of S-BAC (Section 3.3), these attacks do not reply on any racing conditions and therefore are effective even under asynchrony.

	Phase 2 of S-BAC		
	Shard 1	**Shard 2**	**Shard 3** (potential victim)
1	accept(T)	accept(T)	-
	create y_1; inactivate x_1	create y_2; inactivate x_2	create y_3
2	$\triangleright$accept(T)		create y_3
3		$\triangleright$accept(T)	create y_3
4	$\triangleright$accept(T)	$\triangleright$accept(T)	create y_3
5	abort(T)	abort(T)	-
	(unlock x_1)	(unlock x_2)	-
6	$\triangleright$accept(T)		create y_3
7		$\triangleright$accept(T)	create y_3
8	$\triangleright$accept(T)	$\triangleright$accept(T)	create y_3

Table 2: List of replay attacks against the second phase of S-BAC for all possible executions of the transaction $T(x_1,x_2) \rightarrow (y_1,y_2,y_3)$ as described in Section 2.2. The highlighted rows indicate correct executions of S-BAC (*i.e.*, without the attacker), and the other rows indicate incorrect executions due to the replay attacks. In multirows, the top sub-rows show the protocol messages emitted by shards, and the bottom sub-rows indicate local shard actions as a result of emitting those messages. For example, (column 1, row 1) means that *shard* 1 emits accept(T) (top sub-row), and creates a new object y_1 and inactivates x_1 (bottom sub-row). The first two columns indicate the messages emitted by each shard at the end the second phase of S-BAC, and the last column shows the effect of these messages on the output *shard* 3. Replayed messages are marked with the symbol $\triangleright$. For example $\triangleright$accept(T) at (column 1, row 2) means that the attacker sends to other relevant shards (in this case *shard* 3) a prerecorded accept(T) message impersonating *shard* 1.

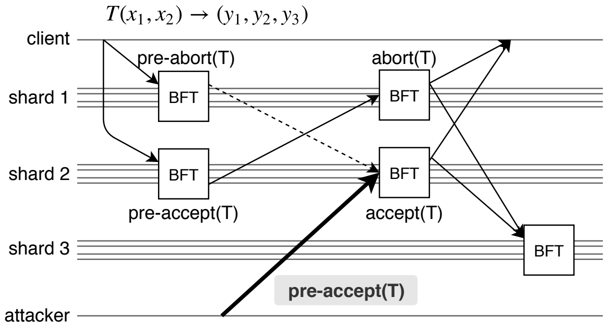

Figure 3: Illustration of the replay attack depicted in row 6 of Table 1. The attacker replays to *shard* 2 a prerecorded pre-accept(T) message (shown as a bold line) from *shard* 1, which precludes *shard* 1's pre-abort(T) message (shown as a dotted line).

3.5 Real-world Impact

The real-world impact and attacker incentives to conduct these attacks depends on the nature and implementation of the smart contract handling the target objects. We discuss the impact of these attacks in the context of two common smart contract applications, which are also described in the Chainspace paper [1]. To take a concrete example, we illustrate the attack depicted in row 3 of Table 1, but similar results can be obtained with the other attacks described in Table 1 and Table 2.

One of the most common blockchain application is to manage cryptocurrency (or coins) and enable payments for processing transactions, implemented by the CSCoin smart contract in Chainspace. Lets suppose object x_1 (handled by *shard* 1) represents Alice's account, and object x_2 (handled by *shard* 2) represents Bob's account. To transfer v coins to Bob, Alice submits a transaction $T(x_1,x_2) \rightarrow (y_1,y_2)$, where y_1 and y_2 respectively represent the new account objects of Alice and Bob, with updated account balances. By executing the attack described in row 3 of Table 1, an attacker can trick *shard* 1 to abort the transaction and unlock x_1 (thus reestablishing Alice's account balance as it was prior to the coin transfer), and *shard* 2 to accept the transaction and create y_2 (thus adding v coins to Bob's account). This attack effectively allows any attacker to double-spend coins on the ledger; and shows how to create v coins out of thin air.

Another common blockchain use case is a platform for decision making (e-voting), implemented by the SVote smart contract in Chainspace. Upon initialization, the SVote contract creates two objects: (*i*) x_1 representing the tally's public key, a list of all voters' public keys, and the tally's signature on these; and (*ii*) x_2 representing a vote object at the initial stage of the election (all candidates having a score of zero) along with a zero-knowledge proof asserting the correctness of the initial stage. To vote, clients submit a transaction $T(x_1,x_2) \rightarrow (y_1,y_2)$, where y_1 and y_2 are respectively the updated voting list (*i.e.*, the voting list without the client's public key), and the election stage updated with the client's vote. By executing the attack described by row 3 of Table 1, an attacker can trick *shard* 1 to abort the transaction and thus not update the voting list, and *shard* 2 to accept the transaction and thus update the election stage. This effectively allows any client to vote multiple times during an election while remaining undetected (due to the privacy-preserving properties of the smart contract).

4 Client-led Cross-shard Consensus Protocols

We describe replay attacks on client-led cross-shard consensus protocols. We illustrate these attacks in the context of Omniledger [6] (Section 4.1) to make the discussion concrete. However, we note that these attacks can be generalised to other similar systems. We discuss how the attacker can record shard messages to replay in future attacks (Section 4.2). In Sections 4.3 and 4.4, we describe replay attacks on the first and second phase of the cross-shard consensus protocol. Finally, we discuss the real-world impact of these attacks (Section 4.5).

4.1 Omniledger Overview

Omniledger uses a client-led cross-shard consensus protocol called Atomix. The client submits the transaction T to the input shards. Each shard runs a BFT protocol locally to decide whether to accept or reject the transaction, and communicates its response (pre-accept(T) or pre-abort(T)) to the client.[1] A shard emits pre-abort(T) if the transaction fails local checks. Alternatively, if a shard emits pre-accept(T), it inactivates the input objects it manages. This is the first phase of Atomix, and is similar to the voting phase in the two-phase atomic commit protocol (Section 2.1), but differs in that the protocol proceeds optimistically. The write changes made by the input shards in the first phase of Atomix are considered permanent (*i.e.*, there is no 'locked' object state), unless the client requests the input shards to revert their changes in the second phase.

After the client has collected pre-accept(T) from all input shards, it submits accept(T) message (containing proof of the pre-accept(T) messages) to the output shards which create the output objects. Alternatively, if any of the input shards emits pre-abort(T), the client sends abort(T) (containing proof of pre-abort(T)) to the relevant input shards which make the input objects active again. This is the second phase of Atomix, and is similar to the commit phase in the two-phase atomic commit protocol (Section 2.1).

Figure 4 shows execution of Atomix for a valid transaction $T(x_1,x_2) \rightarrow (y_1,y_2,y_3)$, with two active inputs (x_1 managed by *shard* 1, and x_2 managed by *shard* 2) and producing three outputs (y_1,y_2,y_3) managed by *shard* 1, *shard* 2 and *shard* 3, respectively. The client sends T to the input shards, both of which reply with pre-accept(T) and make the input objects x_1 and x_2 inactive. The client then sends accept(T) to the output shards which respectively create objects y_1, y_2, and y_3.

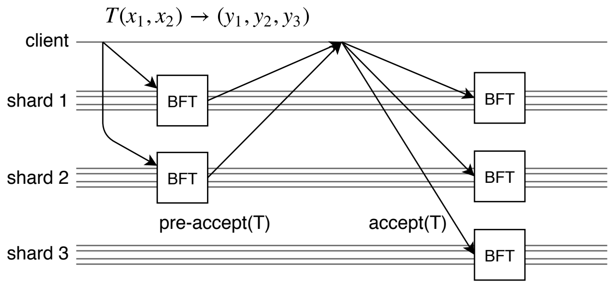

Figure 4: An example execution of Atomix for a valid transaction $T(x_1,x_2) \rightarrow (y_1,y_2,y_3)$ with two inputs (x_1 and x_2, both are active) and three outputs (y_1,y_2,y_3), where the final decision is accept(T).

4.2 Message Recording

Before launching the replay attacks, the attacker first records the target shard responses. The attacker can record shard responses in the first phase of Atomix (*i.e.*, pre-accept(T) or pre-abort(T)), enabling the attacks described in Section 4.3. The attacker can also record shard responses in the second phase of Atomix (*i.e.*, accept(T) or abort(T)), enabling the attacks described in Section 4.4.

In the general case, the attacker passively collects the messages to replay, for example by sniffing on protocol executions, or by downloading the blockchain and selecting the messages to replay.

Eliciting messages to replay. The attacker can act as (or collude with) a client to actively elicit and record the target messages to later use in the replay attacks. As a result, the attacker is able to actively orchestrate the attacks, instead of passively collecting the messages to replay. We describe how the attacker can trigger target messages with the help of an example, without loss of generality. Lets assume that *shard* 1 manages objects x_1 ('active') and object $\widetilde{x_1}$ ('inactive' or non-existent), and *shard* 2 manages object x_2 ('active'); $\widetilde{x*}$ means any inactive object on the shard, and $y*$ means any output object (*i.e.*, their details do not matter).

To elicit pre-accept(T) from *shard* 1 for a transaction $T(x_1,x_2) \rightarrow (y*)$ (the output $y*$ is not relevant here) from *shard* 1, the key consideration is to closely precede the transaction with another transaction that: (*i*) temporarily spends the inputs managed by at least one other shard (in this case x_2 on *shard* 2); and (*ii*) to ensure that the preceding transaction is ultimately aborted so that x_2 becomes active again. The steps look as follows:

- The attacker submits $T'(x_2,\widetilde{x*}) \rightarrow (y*)$ to *shard* 2, where $\widetilde{x*}$ is managed by a different shard. *Shard* 2 emits pre-accept(T') and marks x_2 as inactive.

- The attacker follows up by submitting $T(x_1,x_2) \rightarrow (y*)$ to *shard* 1 and *shard* 2. *Shard* 1 generates pre-accept(T), which is the target message that the attacker

---

[1] For consistency and clarity, we use the terminology introduced in Section 2. In Omniledger, pre-accept(T) is actually a *proof-of-accept* and pre-abort(T) is a *proof-of-abort* [6].

records. *Shard* 2 generates pre-abort(T) because x_2 is inactive.

- The attacker submits abort(T) to *shard* 1 to reactivate x_1, and sends abort(T') to *shard* 2 to reactivate x_2.

For the attacks described in Section 4.4, the attacker needs to elicit abort(T) and accept(T) from the target shards. For the former, the attacker can follow the steps described previously to elicit pre-accept(T) and pre-abort(T). To elicit accept(T), the attacker simply submits transaction T and observes and records its successful execution.

4.3 Attacks on the First Phase of Atomix

We present replay attacks on the first phase of Atomix by taking the example of a transaction $T(x_1,x_2) \rightarrow (y_1,y_2,y_3)$ as described in Section 2.2. These attacks easily generalise to transactions with k inputs and k' outputs managed by an arbitrary number of shards. The replay attacks work in two steps: (*i*) the attacker observes the traffic and records pre-accept(T) or pre-abort(T) messages as described in Section 4.2; and (*ii*) then replay those messages.

Table 3 shows the replay attacks that the attacker can launch, for all possible combinations of responses generated by *shard* 1 and *shard* 2 in the first phase of Atomix. The caption includes details about how to interpret this table. We describe row 2 of Table 3, to help readers interpret rest of the table on their own. In the correct execution (row 1), both *shard* 1 and *shard* 2 emit pre-accept(T) in the first phase, and inactivate the input objects x_1 and x_2. Upon receiving these messages, the client sends accept(T) to the output shards *shard* 1, *shard* 2 and *shard* 3, which create the output objects y_1, y_2 and y_3, respectively; and the protocol terminates. In the attack illustrated in row 2 of Table 3, the attacker races *shard* 1 by sending to the client the prerecorded pre-abort(T) message from *shard* 1. As a result, the client sends abort(T) message to the input shards *shard* 1 and *shard* 2, which re-activate the input objects x_1 and x_2. This results in inconsistent state because the output objects (y_1, y_2 and y_3) have been created, while the input objects (x_1 and x_2) are still active—in a correct execution all transaction inputs should be inactivated, and all outputs should be created.

Table 3 shows that through careful selection of the messages to replay, the attacks can be effective against any shard. The attacks illustrated in row 2, row 3, and row 4 only affect availability, while the other attacks compromise consistency (*i.e.*, the attacker can trick the input shards to reactivate arbitrary objects, and trick the output shards into creating new objects in violation of the protocol). The potential victims of these attacks include the client (*e.g.*, when the attacker replays the shard messages to it in the first phase of Atomix) and any input or output shards.

4.4 Attacks on the Second Phase of Atomix

We present replay attacks on the second phase of Atomix. The attacker prerecords accept(T) and abort(T) messages as described in Section 4.2.

Table 4 shows replay attacks corresponding to the messages emitted by the client in the second phase—*i.e.*, accept(T) in row 1, or abort(T) in row 3. The caption includes details about how to interpret this table. The abort(T) message at (column 1, row 2) means that the attacker sends a prerecorded abort(T) message to the input shards (*shard* 1 and *shard* 2) impersonating the client. Upon receiving this message, *shard* 1 and *shard* 2 (incorrectly) re-activate x_1 and x_2, respectively. Furthermore, all output shards create the output objects when the correct accept(T) message emitted by the client (row 1, column 1) reaches them. This results in inconsistent state, because the output objects have been created, but the input objects have not been consumed and have been reactivated by the abort(T) message replayed by the adversary. The potential victims of abort(T) replay attack are the input shards.

Similarly, accept(T) at (row 4, column 1) means that the attacker sends a prerecorded accept(T) message to the output shards (*shard* 1, *shard* 2 and *shard* 3) impersonating the client. Upon receiving this message, the output shards (incorrectly) create y_1, y_2 and y_3. Furthermore, the input shards (*shard* 1 and *shard* 2) reactivate x_1 and x_2 upon receiving the the correct abort(T) message emitted by the client (row 3, column 1). This creates inconsistent state: the input objects have not been consumed and have been reactivated by the abort(T) message emitted by the client, but the output objects have been created due to the accept(T) message replayed by the attacker. The potential victims of accept(T) replay attack are the output shards.

These attacks are possible because output shards create objects directly upon receiving accept(T); they do not check if the objects have been previously invalidated because shards do not keep records of inactive objects (per the UTXO model) for scalability reasons.[2] The attacker can double-spend the output objects repeatedly from a single prerecorded message by replaying it multiple times, and spending the object (and effectively purging it from the output shards' UTXO) before each replay.

Similar to the attacks against the second phase of S-BAC (Section 3.4), these attacks do not exploit any racing condition and are therefore effective even under asynchrony.

4.5 Real-world Impact

Contrarily to Chainspace, Omniledger does not support smart contracts and only handles a cryptocurrency. The at-

---

[2] Verifying that objects have not been previously invalided implies either keep a forever-growing list of invalidated objects, or download and check the shard's entire blockchain.

	Phase 1 of Atomix			Phase 2 of Atomix		
	Shard 1 (potential victim)	**Shard 2** (potential victim)	**Client** (victim)	**Shard 1** (potential victim)	**Shard 2** (potential victim)	**Shard 3** (potential victim)
1	pre-accept(T) inactivate x_1	pre-accept(T) inactivate x_2	accept(T)	- create y_1	- create y_2	- create y_3
2	▷ pre-abort(T)		abort(T)	- re-activate x_1	- re-activate x_2	-
3		▷ pre-abort(T)	abort(T)	- re-activate x_1	- re-activate x_2	-
4	▷pre-abort(T)	▷pre-abort(T)	abort(T)	- re-activate x_1	- re-activate x_2	-
5	pre-abort(T) -	pre-accept(T) inactivate x_2	abort(T)	- -	- re-activate x_2	-
6	▷pre-accept(T)		accept(T)	- create y_1	- create y_2	- create y_3
7	pre-accept(T) inactivate x_1	pre-abort(T) -	abort(T)	- re-activate x_1	- -	-
8		▷ pre-accept(T)	accept(T)	- create y_1	- create y_2	- create y_3
9	pre-abort(T) -	pre-abort(T) -	abort(T)	- -	- -	-
10	▷ pre-accept(T)	▷ pre-accept(T)	accept(T)	- create y_1	- create y_2	- create y_3

Table 3: List of replay attacks against the first phase of Atomix for all possible executions of the transaction $T(x_1,x_2) \rightarrow (y_1,y_2,y_3)$ as described in Section 2.2. The highlighted rows indicate correct executions of Atomix (*i.e.*, without the attacker), and the other rows indicate incorrect executions due to the replay attacks. In multirows, the top sub-rows show the protocol messages emitted by shards, and the bottom sub-rows indicate local shard actions as a result of emitting those messages. For example, (column 1, row 1) means that *shard* 1 emits pre-accept(T) (top sub-row), and inactivates x_1 (bottom sub-row). The first two columns indicate the messages emitted by each shard at the end the first phase of Atomix. Replayed messages are marked with the symbol ▷, for example ▷pre-abort(T) at (column 1, row 2) means that the attacker sends to the client a prerecorded pre-abort(T) message impersonating *shard* 1 that races the original pre-accept(T) (column 1, row 1) emitted by *shard* 1. The third column indicates the messages sent by the client to the relevant shards, and the last three columns indicate the local actions performed by shards at the end of the second phase of Atomix.

tacks described in Sections 4.3 and 4.4 allow an attacker to: (*i*) double-spend the coins of any user, by reactivating spent coins (*e.g.*, the attacker may execute the attack depicted by row 2 of Table 4 to re-activate the objects x_1 and x_2 after the transfer is complete); and (*ii*) create coins out of thin air by replaying the message to create coins (*e.g.*, an attacker may execute the attack depicted by row 4 of Table 4 to create multiple times object y_3, by purging it from the UTXO list of *shard* 3 prior to each instance of the attack). If the attacker colludes with the client, it can trigger the prerecorded messages needed for the attacks as described in Section 4.2. Alternatively, the attacker can passively observe the network and collect the target messages to replay. Similar results can be obtained using the attacks described in Table 3.

Note that since transaction are recorded on the blockchain, these attacks can be detected retrospectively. This can lead to the attacker being exposed, or the attacker can inculpate innocent users (since the attacker can replay messages of any user).

5 The Byzcuit Atomic Commit Protocol

We previously discussed the two main approaches to achieve cross-shard consensus in sharded blockchains: shard-led protocols in the context of S-BAC (Section 3.1), and client-led protocols in the context of Atomix (Section 4.1). S-BAC runs the protocol among the shards, without relying on client coordination. But this comes at the cost of increased cross-shard communication: all input shards communicate with all other input shards, which leads to communication complexity of $O(n^2)$ where n is the number of input shards.

On the other hand, Atomix is a simpler protocol, and using the client to coordinate cross-shard communication can

		Phase 2 of Atomix		
	Client	**Shard 1** (potential victim)	**Shard 2** (potential victim)	**Shard 3** (potential victim)
1	accept(T)	- create y_1	- create y_2	- create y_3
2	▷abort(T)	- re-activate x_1	- re-activate x_2	-
3	abort(T)	- re-activate x_1	- re-activate x_2	-
4	▷accept(T)	- create y_1	- create y_2	- create y_3

Table 4: List of replay attacks against the second phase of Atomix for all possible executions of the transaction $T(x_1,x_2) \rightarrow (y_1,y_2,y_3)$ as described in Section 2. The highlighted rows indicate correct executions of Atomix (*i.e.*, without the attacker), and the other rows indicate incorrect executions due to the replay attacks. In multirows, the top sub-rows show the protocol messages emitted by shards, and the bottom sub-rows indicate local shard actions. Note that we use the multirow format for consistency reasons; in this table the first column indicates the messages emitted by the client at the beginning of the second phase of Atomix, and the last two column shows the effect of these messages on the relevant shards. Replayed messages are marked with the symbol ▷. For example, ▷abort(T) at (column 1, row 2) means that the attacker sends a prerecorded abort(T) message to the input shards impersonating the client.

reduce the cost to $O(n)$ in the number of shards (by aggregating shard messages). However, an unresponsive or malicious client can permanently lock input objects by never initiating the second phase of the protocol, requiring additional design considerations (*e.g.*, a new entity that periodically unlocks input objects for transactions on which no progress has been made). Moreover, we have highlighted that both shard-led (Sections 3.3 and 3.4) and client-led (Sections 4.3 and 4.4) protocols are vulnerable to replay attacks that can compromise system liveness and safety.

Motivated by these insights, we present Byzcuit—a cross-shard atomic commit protocol that is based on S-BAC, and integrates design features from Atomix. Byzcuit allocates a Transaction Manager (TM) to coordinate cross-shard communication, reducing its cost to $O(n)$ in the happy case[3]; alternatively Byzcuit also has a fall-back mode in case the TM fails, similar to Atomix and traditional two phase commit protocols. Byzcuit achieves resilience against the replays attacks described in Section 3 and Section 4, by leveraging the notion of dummy objects and object sequence numbers, which have been explained in the following subsections.

5.1 Byzcuit Protocol Design

We first describe the main ingredients of our solution to the replay attacks. We observe that the replay attacks described in Section 3 and Section 4 are possible because of two reasons. First, the input shards do not have a way to know that the protocol messages that they receive correspond to which instance (or session) of a transaction. This gap in the input shards' knowledge enables an attacker to replay old messages. To address this limitation, we associate a session identifier with each transaction. The second reason that facilitates the replay attacks is that in some cases the output shards are only involved in the second phase of the protocol, and therefore have no knowledge of the transaction context (to determine freshness) that is available to the input shards. We address this limitation by introducing the notion of *dummy objects*. Each shard creates a fixed number of dummy objects upon configuration. If a shard only serves as an output shard for a transaction (and therefore will only be involved in the second phase of the protocol), Byzcuit forces it to be involved in the first phase of the protocol by implicitly including a dummy object managed by the output shard in the transaction inputs, which will create a new dummy object upon completion. As a result, the output shard also becomes the input shard (because of the inclusion of its dummy object in the transaction inputs) and witnesses the entire protocol execution, rather than just the second phase.

Byzcuit Protocol Execution. We illustrate Byzcuit taking the example of a transaction $T(x_1,x_2) \rightarrow (y_1,y_2,y_3)$ with two input objects, x_1 managed by *shard* 1 and x_2 managed by *shard* 2; and three outputs, y_1 managed by *shard* 1, y_2 managed by *shard* 2, and y_3 managed by *shard* 3.

Figure 5 illustrates the Byzcuit protocol; the client first sends the transaction to all input and output shards. Note that this is different than other protocols like S-BAC and Atomix, where the transaction is only sent to the input shards. As mentioned previously, to achieve resilience against replay attacks, Byzcuit forces a shard that is *only* involved in creating the output objects to also become an input shard (and witness the transactional context by participating in the first phase of the protocol) by implicitly consuming one of its dummy inputs (which creates a new dummy object upon completion). Byzcuit associates a sequence number s_{x_i} to each object and

---

[3] The communication complexity can be reduced to $O(n)$ in the number of shards by aggregating shard messages as described by Omniledger.

dummy object (when the object is created $s_{x_i} = 0$). The sequence number is intrinsically linked to the object: when clients query shards to obtain an object x_i, they also receive the associated sequence number s_{x_i}.

When submitting the transaction T, the client also sends along a transaction sequence number $s_T = max\{s_{x_1}, s_{x_2}, s_{d_3}\}$, where the transaction sequence number s_T is the maximum of the sequence numbers s_{x_i} of each input object x_i and dummy objects d_i (❶).

Upon receiving a new pair (T, s_T), each shard saves (T, s_T) in a local cache memory—the transaction sequence number s_T acts as session identifier associated with the transaction T. Each shard internally verifies that the transaction passes local checks, and that s_T is equal to (or bigger than) the sequence numbers of the objects they manage (*i.e.*, *shard* 1 checks $s_T \geq s_{x_1}$, *shard* 2 checks $s_T \geq s_{x_2}$, *shard* 3 checks $s_T \geq s_{d_3}$). The shards send their local decision to the TM: pre-accept(T, s_T) for local accept (accompanied by the shard locking the objects it manages), or pre-abort(T, s_T) for local abort.

After receiving all the messages corresponding to the first phase of Byzcuit from the concerned shards, the TM sends a suitable message to the shards (accept(T, s_T) if all the shards respond with pre-accept(T, s_T), or abort(T, s_T) otherwise). Upon receiving accept(T, s_T) or abort(T, s_T) from the TM, shards first verify that they previously cached the pair (T, s_T) associated with the message; otherwise they ignore it (❷).

The accept(T, s_T) or abort(T, s_T) messages sent by the TM provide enough evidence to the shards to verify whether s_T is correctly computed; that is, shards verify that s_T is at least the maximum of the sequence numbers of each input and dummy object by inspecting the transaction T signed by each shard. If the accept(T, s_T) message has a correct s_T, the shards inactivate the input objects and create the output objects (y_1, y_2, y_3), and *shard* 3 creates a new dummy object $\widetilde{d_3}$; otherwise, they update the sequence numbers of each input object (s_{x_1}, s_{x_2}) and dummy object d_3 to $(s_T + 1)$, *i.e.* shards locally update $s_{x_1} \leftarrow (s_T + 1)$ and $s_{x_2} \leftarrow (s_T + 1)$, and $s_{d_3} \leftarrow (s_T + 1)$. Shards delete (T, s_T) from their local cache (❸).

As we assume that shards are honest, it suffices if only one shard notifies the client of the protocol outcome; we may set any arbitrary rule to decide which shard notifies the client (*e.g.*, the shard handling the first input object) (❹).

Figure 6 shows the finite state machine describing the life cycle of Byzcuit objects.

Transaction Manager. The Transaction Manager (TM) coordinates cross-shard communication in Byzcuit. We now discuss who might play the role of the TM, and argue that Byzcuit guarantees liveness even if the TM is malicious.

Keeping with the overall design goal of decentralization, we envision that a designated shard will act as the TM. If the shard is honest, the TM is live—and therefore progress is always made. The input shards contact in turn each node of the TM shard until they reach one honest node. The TM shard may have up to f dishonest nodes; therefore, the client or the input shards need to send messages to at least $f+1$ nodes of the TM shard to ensure that it is received by at least one honest node. Thus, as soon as the first honest node receives the message, the protocol progresses.

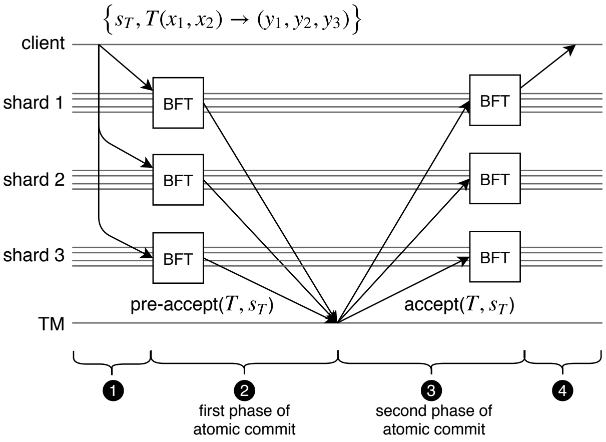

Figure 5: An example execution of Byzcuit for a valid transaction $T(x_1, x_2) \rightarrow (y_1, y_2, y_3)$ with two input objects (x_1 and x_2, both are active), and three outputs (y_1, y_2, y_3), where the final decision is accept(T, s_T).

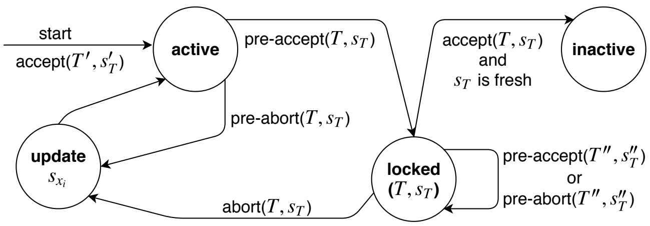

Figure 6: State machine representing the life cycle of objects in Byzcuit. Objects are initially 'active'. Upon receiving a transaction that passes local checks, a shard changes its input objects' state to 'locked' (objects are locked for a given transaction T and transaction sequence number s_T) and emits pre-accept(T, s_T); otherwise it updates the sequence number of every object it manages and emits abort(T, s_T). Once a shard locks input objects for a given (T, s_T), any accept(T, s_T) and abort(T, s_T) with malformed transaction sequence numbers are ignored, and do not cause any transition (not included in the figure). Any incoming transaction T' that requires processing 'locked' input object(s) is aborted. Upon receiving accept(T, s_T) with a well formed s_T, a shard makes its input objects 'inactive' and creates the output objects. Alternatively, upon receiving abort(T, s_T) with a well formed s_T a shard unlocks its input objects and updates the corresponding sequence numbers.

If the TM is the client or any centralized party, it may act arbitrarily—but this does not stall the protocol because anyone can make the protocol progress by taking over at any time the role of the TM. This is possible because the TM does not act on the basis of any secrets, therefore anyone else can take over and complete the protocols. This "anyone" may be an honest node in a shard that wants to finally unlock an object (*e.g.*, upon a timeout); or other clients that wish to use a locked object; or it may be an external service that has a job to periodically close open Byzcuit instances. Therefore, Byzcuit guarantees liveness as long as there is at least one honest entity in the system.

5.2 Security against Replay Attacks

We argue that Byzcuit is resilient to replay attacks. We recall the Honest Shard assumption from Chainspace under which Byzcuit operates, and assume that messages are authenticated as in traditional BFT protocols.

Security Assumption 1. *(Honest Shard [1]) The adversary may create arbitrary smart contracts, and input arbitrary transactions into Byzcuit, however they are bound to only control up to f faulty nodes in any shard. As a result, and to ensure the correctness and liveness properties of Byzantine consensus, each shard must have a size of at least $3f+1$ nodes. (From Chainspace [1].)*

Any message emitted by shards comes with at least $f+1$ signatures from nodes. Assuming honest shards, the attacker can forge at most f signatures, which is not enough to impersonate a shard.

Security of the first phase of Byzcuit. An attacker may try to replay $\mathsf{pre\text{-}accept}(T, s_T)$ and $\mathsf{pre\text{-}abort}(T, s_T)$ messages during the first phase of the protocol, similarly to the attacks described in Sections 3.3 and 4.3; the TM then aggregates these messages into either $\mathsf{accept}(T, s_T)$ or $\mathsf{abort}(T, s_T)$, and forwards them to the shards during the second phase of the protocol. Theorem 1 shows that Byzcuit detects that they originate from replayed messages and ignores them.

Theorem 1. *Under Honest Shard assumption, Byzcuit ignores $\mathsf{accept}(T, s_T)$ and $\mathsf{abort}(T, s_T)$ messages issued from replayed $\mathsf{pre\text{-}accept}(T, s_T)$ and $\mathsf{pre\text{-}abort}(T, s_T)$.*

A proof of Theorem 1 can be found in Appendix B.1. Intuitively, the transaction sequence number s_T acts as session identifier associated with the transaction T; the attacker cannot obtain prerecorded messages containing a fresh s_T. Byzcuit shards can then distinguish replayed messages (*i.e.*, messages with old s_T) from the messages coming from the instance of the protocol that they are executing (*i.e.*, messages with fresh s_T).

Security of the second phase of Byzcuit. An attacker may try to replay $\mathsf{accept}(T, s_T)$ and $\mathsf{abort}(T, s_T)$ messages during the second phase of the protocol, similarly to the attacks described in Sections 3.4 and 4.4; Theorem 2 shows that Byzcuit ignores those replayed messages.

Theorem 2. *Under Honest Shard assumption, Byzcuit ignores replayed $\mathsf{accept}(T, s_T)$ and $\mathsf{abort}(T, s_T)$ messages.*

A proof of Theorem 2 can be found in Appendix B.2. Intuitively, these attacks target shards acting only as output shards (and not also as input shards) and exploit the fact that they are only involved in the second phase of the protocol, and therefore have no knowledge of the transaction context (to determine freshness) that is available to the input shards. Byzcuit is resilient to these replay attacks as it is designed in such a way that there are no shards that act only as output shards; all output shards are forced to also become input shards, by introducing dummy objects if they do not manage any input objects; this prevents the attacks as the attack targets no longer exist.

6 Conclusion

We presented the first replay attacks against cross-shard consensus protocols in sharded distributed ledgers. Those attacks affect both shard-driven and client-driven consensus protocols, and allows 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. We present Byzcuit, a new cross-shard consensus protocol merging features from shard-driven and client-driven consensus protocols, and withstanding replay attacks.

Acknowledgements

George Danezis, Shehar Bano and Alberto Sonnino are supported in part by EPSRC Grant EP/N028104/1 and the EU H2020 DECODE project under grant agreement number 732546 as well as `chainspace.io`. Mustafa Al-Bassam is supported by The Alan Turing Institute. We thank Eleftherios Kokoris-Kogias for helpful suggestions on early manuscripts.

References

[1] AL-BASSAM, M., SONNINO, A., BANO, S., HRYCYSZYN, D., AND DANEZIS, G. Chainspace: A Sharded Smart Contracts Platform. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2018).

[2] BANO, S., AL-BASSAM, M., AND DANEZIS, G. The Road to Scalable Blockchain Designs. *;login: The USENIX Magazine 42*, 4 (2017).

[3] BUTERIN, V. Cross-shard contract yanking. https://ethresear.ch/t/cross-shard-contract-yanking/1450, 2018.

[4] CASTRO, M., LISKOV, B., ET AL. Practical byzantine fault tolerance. In *OSDI* (1999), vol. 99, pp. 173–186.

[5] GRAY, J. N. Notes on database operating systems. In *Operating Systems*. Springer, 1978, pp. 393–481.

[6] KOKORIS-KOGIAS, E., JOVANOVIC, P., GASSER, L., GAILLY, N., AND FORD, B. Omniledger: A secure, scale-out, decentralized ledger. *IACR Cryptology ePrint Archive 2017* (2017), 406.

[7] NAKAMOTO, S. Bitcoin: A peer-to-peer electronic cash system.

A Comparison with Mutex-based Cross-shard Consensus Protocols

Mutex-based schemes for cross-shard transactions, such as Ethereum's cross-shard "yanking" proposal [3], find a way to avoid complex cross-shard coordination for transactions that involve objects managed by different shards. The key idea is to require all objects that a transaction reads or writes to be in the same shard (*i.e.*, all locks for a transaction are local to the shard). Cross-shard transactions are enabled by transferring the concerned objects between shards, effectively giving shards a lock on those objects. When *shard* 1 transfers an object to *shard* 2, *shard* 1 includes a transfer "receipt" in its blockchain. A client can then send to *shard* 2 a Merkle proof of this receipt being included in *shard* 1's blockchain, which makes the object active in *shard* 2.

Mutex-based schemes also need to consider replay attacks. Clients can claim the same receipt multiple times, unless shards store information about previously claimed receipts. Naïvely, shards have to store information about all previously claimed receipts permanently. However, two intermediate options with trade-offs have been proposed [3]:

- Shards only store information about receipts for l blocks; so clients can only claim receipts within l blocks, and objects are permanently lost if not claimed within l blocks.
- Shards only store information about receipts for l blocks, and include the root of a Merkle tree of claimed receipts in their blockchain every l blocks. If a receipt is not claimed within l blocks, the client must provide one Merkle proof every l blocks that have passed to show that the receipt has not been previously claimed, in order to claim it. The longer the receipt was not claimed, the greater the number of proofs that are needed to claim a receipt. These proofs need to be also stored on-chain to allow other nodes to validate them.

In contrast, Byzcuit prevents the need for shards to store information about old state (such as inactive objects or old receipts) as shards only need to know the set of active objects they manage, and does not impose a trade-off between the amount of information about old state that needs to be stored and the cost of recovering old state that was held up in an incomplete cross-shard transaction (*i.e.*, an unclaimed receipt).

B Proofs of Theorem 1 and Theorem 2

The security proofs of Theorem 1 and Theorem 2 of Section 5.2 rely on Lemma 1.

Lemma 1. *Under Honest Shard assumption, no attacker can obtain prerecorded messages containing a fresh transaction sequence number* s_T.

Proof. The core idea protecting Byzcuit from these replay attacks is that the attacker can only obtain prerecorded messages associated with old transaction sequence numbers s_T. The transaction sequence number s_T is fresh only if it is at least equal the maximum of the sequence number of all input and dummy objects of the transaction T. Shards update every input and dummy object sequence number upon aborting transactions in such a way that sequence numbers only increase. That is, after emitting $\textsf{pre-accept}(T,s_T)$ or $\textsf{pre-abort}(T,s_T)$, either the sequence number of all input and dummy objects of T are updated to a value bigger than s_T (in case of $\textsf{pre-abort}(T,s_T)$), or the objects are inactivated which prevents any successive transaction to use them as input (in case of $\textsf{pre-accept}(T,s_T)$). It is therefore impossible for the adversary to hold a prerecorded message for a fresh s_T since the only prerecorded messages that the adversary can obtain contain sequence numbers smaller than s_T. ☐

B.1 Proof of Theorem 1

Theorem 1. *Under Honest Shard assumption, Byzcuit ignores* $\textsf{accept}(T,s_T)$ *and* $\textsf{abort}(T,s_T)$ *messages issued from replayed* $\textsf{pre-accept}(T,s_T)$ *and* $\textsf{pre-abort}(T,s_T)$*.*

Proof. Figure 6 shows that once Byzcuit locks objects for a particular pair (T,s_T), the protocol can only progress toward $\textsf{accept}(T,s_T)$ or $\textsf{abort}(T,s_T)$; *i.e.* shards can either accept or abort the transaction T. The attacker aims to trick one or more shards to incorrectly accept or abort T by injecting prerecorded messages during the first phase of Byzcuit; we show that the attacker fails in every possible scenario.

Suppose transaction T should abort (the TM outputs $\textsf{abort}(T,s_T)$), but the attacker tries to trick some shards to accept the transaction. Figure 6 shows that the attacker can only succeed the attack if they gather $\textsf{accept}(T,s_T)$ containing a fresh transaction sequence number s_T. Lemma 1 states that no attacker can obtain prerecorded messages over fresh transaction sequence number s_T; therefore the only messages available to the adversary at this point of the protocol are (at most) $n-1$ $\textsf{pre-accept}(T,s_T)$ and (at most) n $\textsf{abort}(T,s_T)$, where n is the number of concerned shards. This is not enough to form an $\mathsf{accept}(T, s_T)$ message with a fresh transaction sequence number s_T (which is composed of n $\mathsf{pre\text{-}accept}(T, s_T)$); therefore the attacker cannot trick any shard to accept the transaction.

Suppose transaction T should be accepted (the TM outputs $\mathsf{accept}(T, s_T)$ with a fresh S_T), but the attacker tries to trick some shards to abort the transaction. Figure 6 show that Byzcuit does not require a fresh transaction sequence number s_T to abort transactions (the freshness of s_T is only enforced upon accepting a transaction); but shards locked the input and dummy objects of the transaction for the pair (T, s_T) (with fresh s_T), so the attacker needs to gather $\mathsf{abort}(T, s_T)$ containing the same transaction sequence number s_T locked by shards. Lemma 1 shows that the attacker cannot obtain prerecorded messages over fresh s_T; therefore the only messages available to the adversary containing the (fresh) s_T locked by shards at this point of the protocol are n $\mathsf{pre\text{-}accept}(T, s_T)$. This is not enough to form an $\mathsf{abort}(T, s_T)$ message (which is composed of at least one $\mathsf{pre\text{-}abort}(T, s_T)$); therefore the attacker cannot trick any shard to abort the transaction. ☐

B.2 Proof of Theorem 2

Theorem 2. *Under Honest Shard assumption, Byzcuit ignores replayed* $\mathsf{accept}(T, s_T)$ *and* $\mathsf{abort}(T, s_T)$ *messages.*

Proof. Figure 6 shows that shards only act upon $\mathsf{accept}(T, s_T)$ and $\mathsf{abort}(T, s_T)$ messages if they have the pair (T, s_T) saved in their local cache[4]. Shards save a pair (T, s_T) in their local cache upon emitting $\mathsf{pre\text{-}accept}(T, s_T)$ or $\mathsf{pre\text{-}abort}(T, s_T)$, and delete it at the end of the protocol; therefore the only attack windows where the adversary can replay $\mathsf{accept}(T, s_T)$ and $\mathsf{abort}(T, s_T)$ messages is while the transaction T (associated with s_T) is being processed by the second phase of Byzcuit. This forces the attacker to operates under the same conditions as Theorem 1, which is proven secure in Appendix B.1. ☐

C Security & Correctness of Byzcuit

We show that Byzcuit guarantees liveness, consistency, and validity similarly to S-BAC.

Theorem 3. *(Liveness [1]) Under Honest Shards assumption, a transaction T that is proposed to at least one honest concerned node, eventually results in either being committed or aborted, namely all parties deciding* $\mathsf{accept}(T, s_T)$ *or* $\mathsf{abort}(T, s_T)$*. (From Chainspace [1].)*

Proof. We rely on the liveness properties of the byzantine agreement (shards with only f nodes eventually reach consensus on a sequence), and the broadcast from nodes of shards to all other nodes of shards, channelled through the Transaction Manager. Assuming T has been given to an honest node, it will be sequenced withing an honest shard BFT sequence, and thus a $\mathsf{pre\text{-}accept}(T, s_T)$ or $\mathsf{pre\text{-}abort}(T, s_T)$ will be sent from the $2f+1$ honest nodes of this shard, aggregated into $\mathsf{accept}(T, s_T)$ or $\mathsf{abort}(T, s_T)$, and sent to the $f+1$ nodes of the other concerned shards. Upon receiving these messages the honest nodes from other shards will process the transaction within their shards, and the BFT will eventually sequence it. Thus the user will eventually receive a decision from at least $f+1$ nodes of a shard. ☐

Theorem 4. *(Consistency [1]) Under Honest Shards assumption, no two conflicting transactions, namely transactions sharing the same input will be committed. Furthermore, a sequential executions for all transactions exists. (From Chainspace [1].)*

Proof. A transaction is accepted only if some nodes receive $\mathsf{accept}(T, s_T)$, which presupposes all shards have provided enough evidence to conclude $\mathsf{pre\text{-}accept}(T, s_T)$ for each of them. Two conflicting transaction, sharing an input, must share a shard of at least $3f+1$ concerned nodes for the common object—with at most f of them being malicious. Without loss of generality upon receiving the $\mathsf{pre\text{-}accept}(T, s_T)$ message for the first transaction, this shard will sequence it, and the honest nodes will emit messages for all—and will lock this object until the two phase protocol concludes. Any subsequent attempt to $\mathsf{pre\text{-}accept}(T, s_T)$ for a conflicting T' will result in a $\mathsf{pre\text{-}abort}(T, s_T)$ and cannot yield a accept, if all other shards are honest majority too. After completion of the first $\mathsf{accept}(T, s_T)$ the shard removes the object from the active set, and thus subsequent T' would also lead to $\mathsf{pre\text{-}abort}(T, s_T)$. Thus there is no path in the chain of possible interleavings of the executions of two conflicting transactions that leads to them both being committed. ☐

Theorem 5. *(Validity [1]) Under Honest Shards assumption, a transaction may only be accepted if it is valid according to the smart contract (or application) logic. (From Chainspace [1].)*

Proof. A transaction is committed only if some nodes conclude that $\mathsf{accept}(T, s_T)$, which presupposes all shards have provided enough evidence to conclude $\mathsf{pre\text{-}accept}(T, s_T)$ for each of them. The concerned nodes include at least one shard per input object for the transaction; for any contract logic represented in the transaction, at least one of those shards will be managing object from that contract. Each shard checks the validity rules for the objects they manage (ensuring they are active) and the contracts those objects are part of (ensuring the transaction is valid with respect to the contract logic) in order to $\mathsf{pre\text{-}accept}(T, s_T)$. Thus if all shards say $\mathsf{pre\text{-}accept}(T, s_T)$ to conclude that $\mathsf{accept}(T, s_T)$, all object have been checked as active, and all the contract calls within

---

[4] Contrarily to S-BAC and Atomix, all Byzcuit shards have the pair (T, s_T) in their local cache after as they all participate to the first phase of the protocol.

the transaction have been checked by at least one shard—whose decision is honest due to at most f faulty nodes. If even a single object is inactive or locked, or a single trace for a contract fails to check, then the honest nodes in the shard will emit pre-abort(T, s_T), and the final decision will be abort(T, s_T). $\square$