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

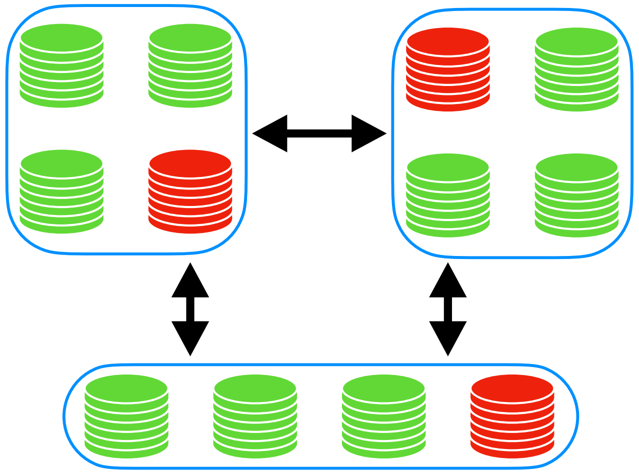
calibra

Blockchains' Scalability

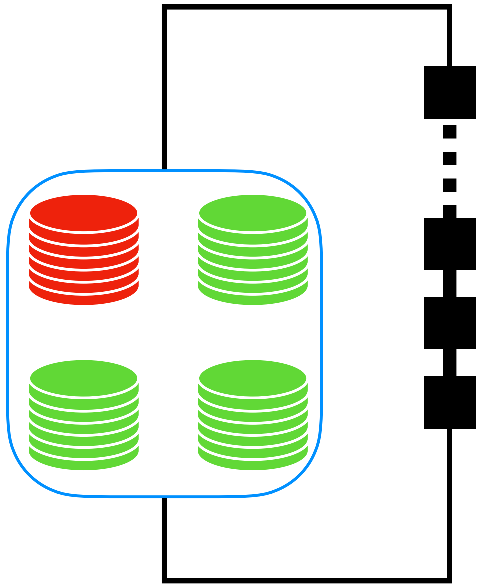
Blockchains' Scalability

- Several ways to enable blockchain scalability

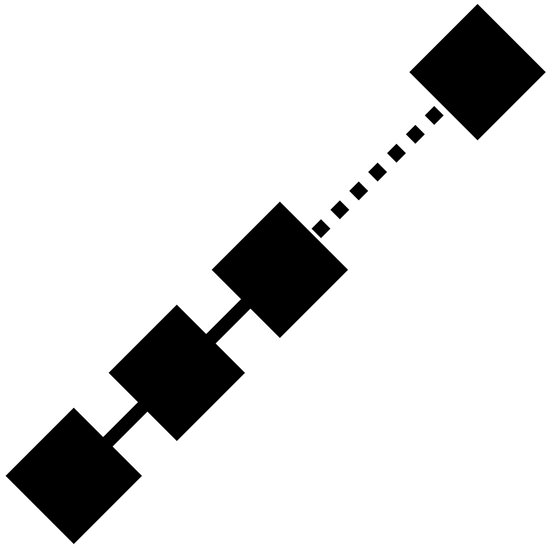
Sharding



State Channels

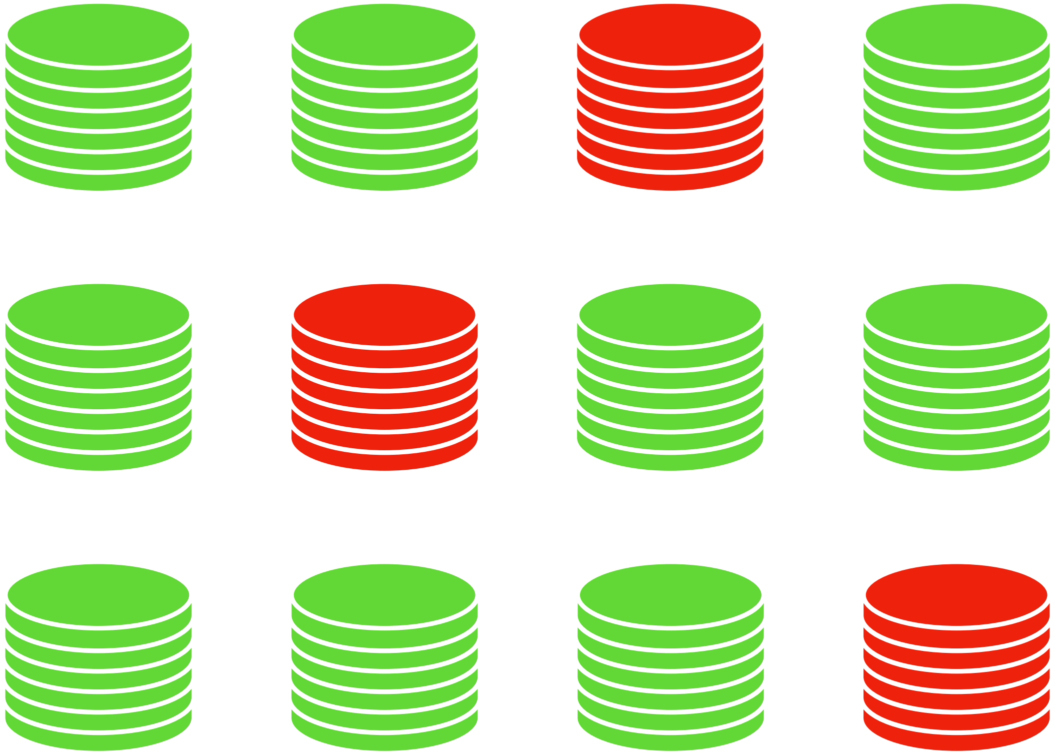


Bigger Blocks



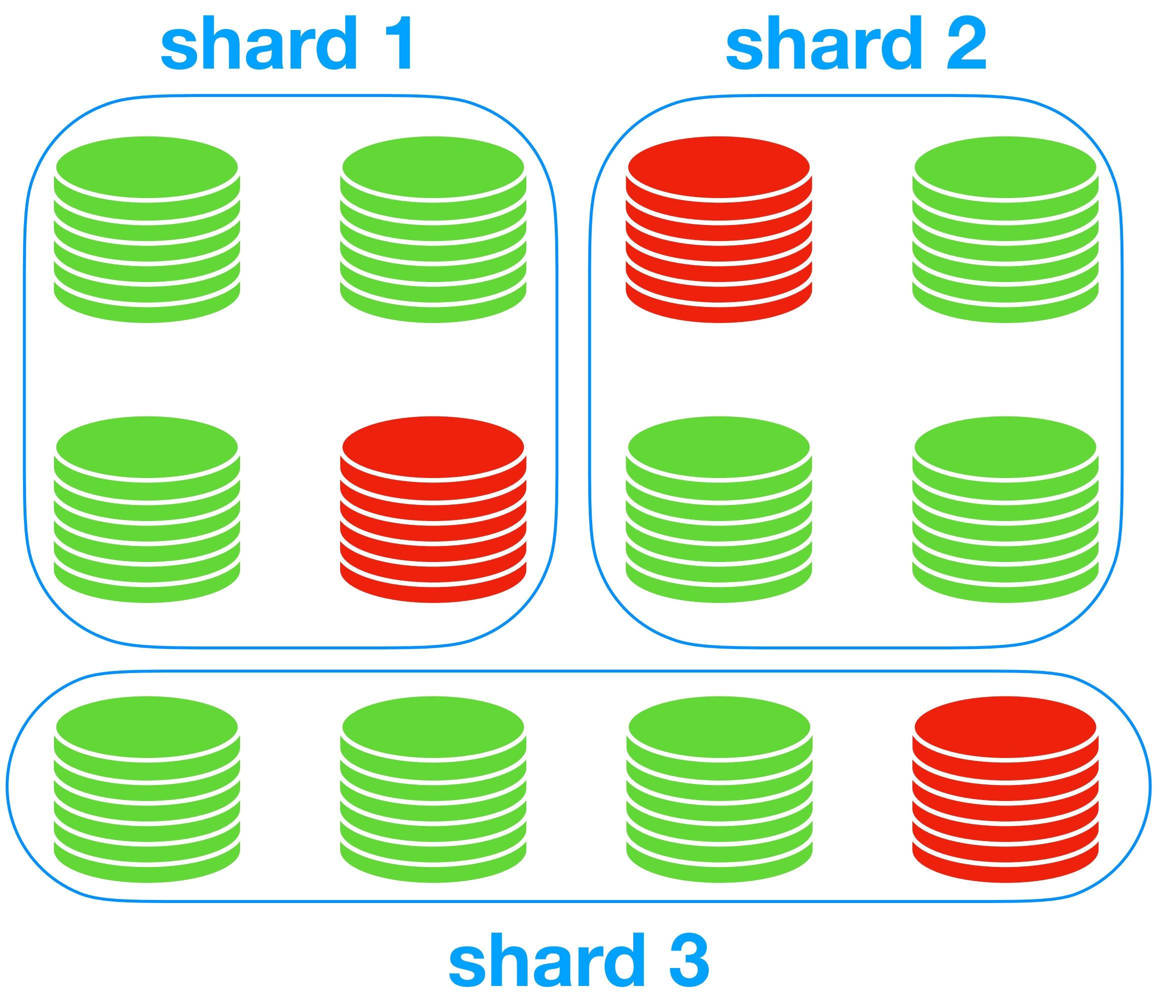
Sharded Distributed Ledgers

- Linear scalability through sharding



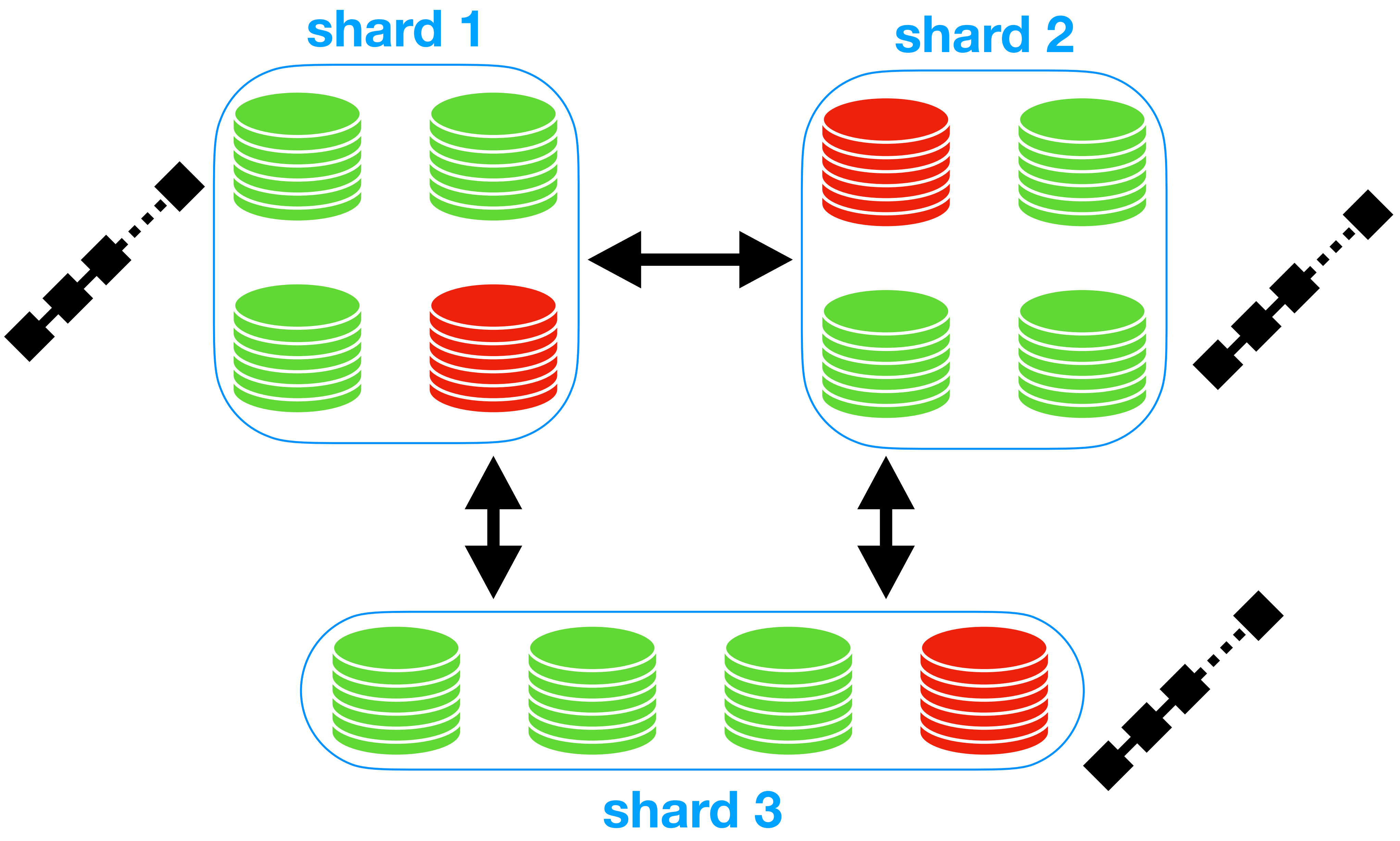
Sharded Distributed Ledgers

- Linear scalability through state sharding



Sharded Distributed Ledgers

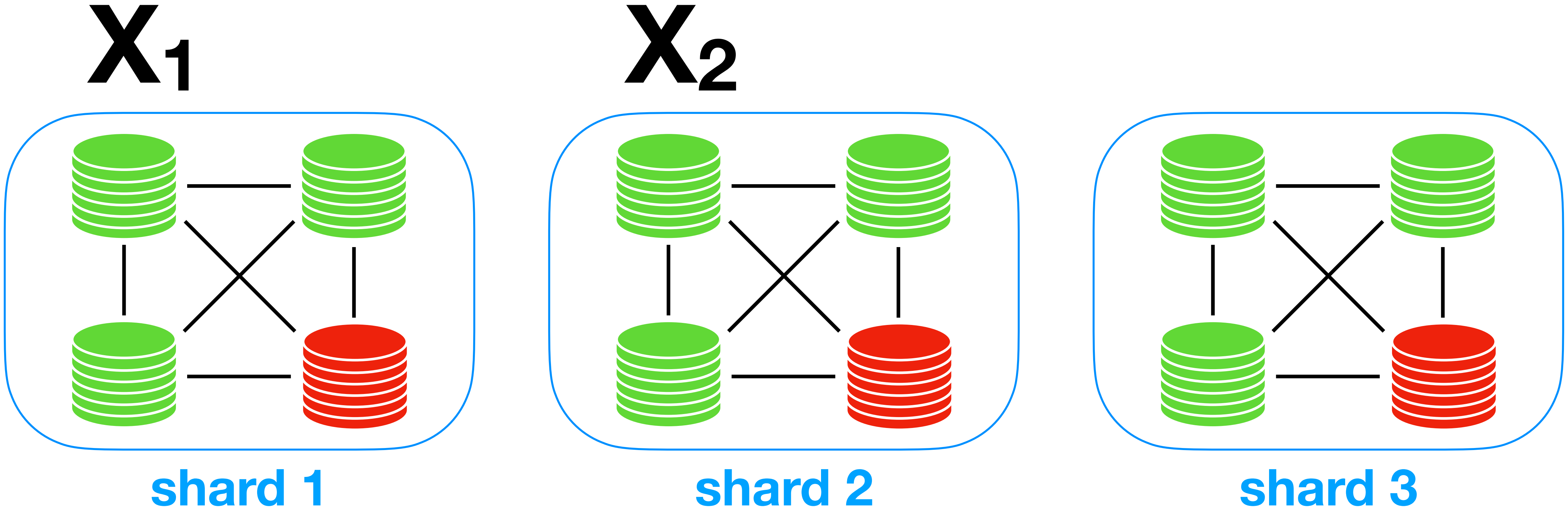
- Linear scalability through state sharding



Sharded Distributed Ledgers

transaction

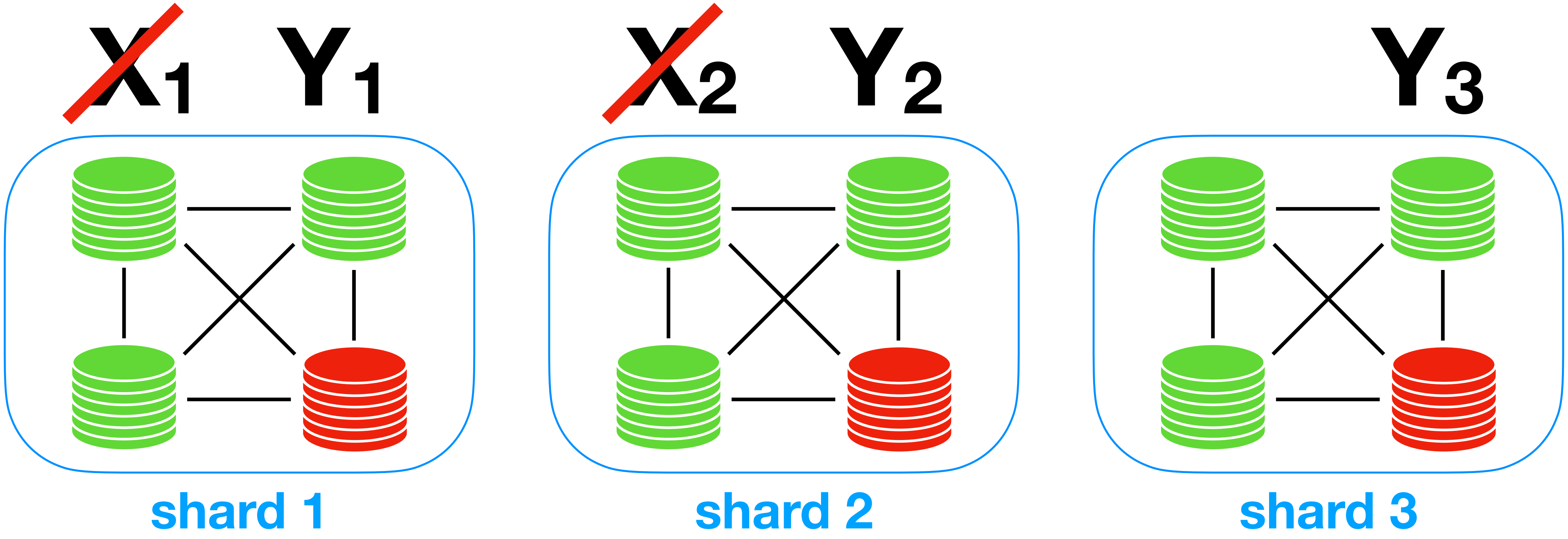
$$T(x_1, x_2) \rightarrow (y_1, y_2, y_3)$$



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 (eg. 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

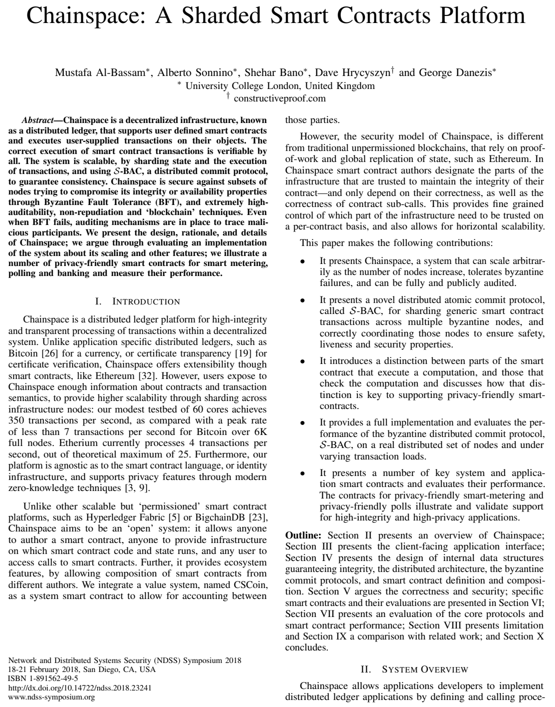
or

Shards store & check old data (break scalability)

Attacks Overview

- Illustration of the attacks

Chainspace



Chainspace: A Sharded Smart Contracts Platform

Mustafa Al-Bassam*, Alberto Sonnino*, Shehar Bano*, Dave Hrycyszyn† and George Danezis*
* University College London, United Kingdom
† constructiveproof.com

***Abstract*—Chainspace is a decentralized infrastructure, known as a distributed ledger, that supports user defined smart contracts and executes user-supplied transactions on their objects. The correct execution of smart contract transactions is verifiable by all. The system is scalable, by sharding state and the execution of transactions, and using $\mathcal{S}$-BAC, a distributed commit protocol, to guarantee consistency. Chainspace is secure against subsets of nodes trying to compromise its integrity or availability properties through Byzantine Fault Tolerance (BFT), and extremely high-auditability, non-repudiation and 'blockchain' techniques. Even when BFT fails, auditing mechanisms are in place to trace malicious participants. We present the design, rationale, and details of Chainspace; we argue through evaluating an implementation of the system about its scaling and other features; we illustrate a number of privacy-friendly smart contracts for smart metering, polling and banking and measure their performance.**

I. Introduction

Chainspace is a distributed ledger platform for high-integrity and transparent processing of transactions within a decentralized system. Unlike application specific distributed ledgers, such as Bitcoin [26] for a currency, or certificate transparency [19] for certificate verification, Chainspace offers extensibility though smart contracts, like Ethereum [32]. However, users expose to Chainspace enough information about contracts and transaction semantics, to provide higher scalability through sharding across infrastructure nodes: our modest testbed of 60 cores achieves 350 transactions per second, as compared with a peak rate of less than 7 transactions per second for Bitcoin over 6K full nodes. Etherium currently processes 4 transactions per second, out of theoretical maximum of 25. Furthermore, our platform is agnostic as to the smart contract language, or identity infrastructure, and supports privacy features through modern zero-knowledge techniques [3, 9].

Unlike other scalable but 'permissioned' smart contract platforms, such as Hyperledger Fabric [5] or BigchainDB [23], Chainspace aims to be an 'open' system: it allows anyone to author a smart contract, anyone to provide infrastructure on which smart contract code and state runs, and any user to access calls to smart contracts. Further, it provides ecosystem features, by allowing composition of smart contracts from different authors. We integrate a value system, named CSCoin, as a system smart contract to allow for accounting between those parties.

However, the security model of Chainspace, is different from traditional unpermissioned blockchains, that rely on proof-of-work and global replication of state, such as Ethereum. In Chainspace smart contract authors designate the parts of the infrastructure that are trusted to maintain the integrity of their contract—and only depend on their correctness, as well as the correctness of contract sub-calls. This provides fine grained control of which part of the infrastructure need to be trusted on a per-contract basis, and also allows for horizontal scalability.

This paper makes the following contributions:

- It presents Chainspace, a system that can scale arbitrarily as the number of nodes increase, tolerates byzantine failures, and can be fully and publicly audited.
- It presents a novel distributed atomic commit protocol, called $\mathcal{S}$-BAC, for sharding generic smart contract transactions across multiple byzantine nodes, and correctly coordinating those nodes to ensure safety, liveness and security properties.
- It introduces a distinction between parts of the smart contract that execute a computation, and those that check the computation and discusses how that distinction is key to supporting privacy-friendly smart-contracts.
- It provides a full implementation and evaluates the performance of the byzantine distributed commit protocol, $\mathcal{S}$-BAC, on a real distributed set of nodes and under varying transaction loads.
- It presents a number of key system and application smart contracts and evaluates their performance. The contracts for privacy-friendly smart-metering and privacy-friendly polls illustrate and validate support for high-integrity and high-privacy applications.

Outline: Section II presents an overview of Chainspace; Section III presents the client-facing application interface; Section IV presents the design of internal data structures guaranteeing integrity, the distributed architecture, the byzantine commit protocols, and smart contract definition and composition. Section V argues the correctness and security; specific smart contracts and their evaluations are presented in Section VI; Section VII presents an evaluation of the core protocols and smart contract performance; Section VIII presents limitation and Section IX a comparison with related work; and Section X concludes.

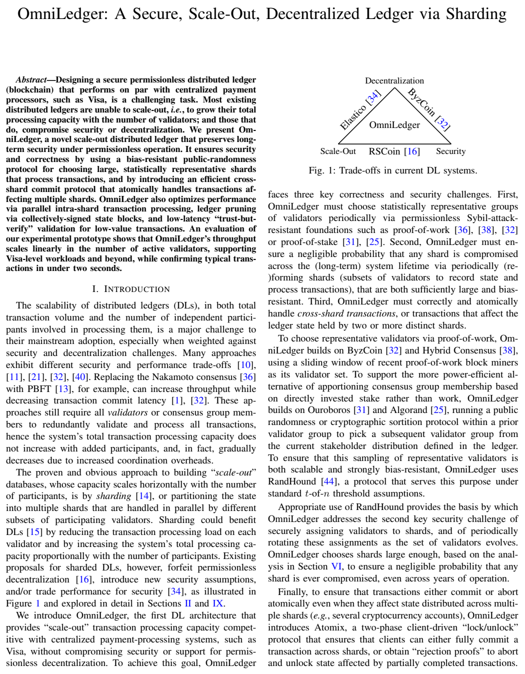
II. System Overview

Chainspace allows applications developers to implement distributed ledger applications by defining and calling proce-

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23241
www.ndss-symposium.org

NDSS'18

Omniledger



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

***Abstract*—Designing a secure permissionless distributed ledger (blockchain) that performs on par with centralized payment processors, such as Visa, is a challenging task. Most existing distributed ledgers are unable to scale-out, *i.e.*, to grow their total processing capacity with the number of validators; and those that do, compromise security or decentralization. We present OmniLedger, a novel scale-out distributed ledger that preserves long-term security under permissionless operation. It ensures security and correctness by using a bias-resistant public-randomness protocol for choosing large, statistically representative shards that process transactions, and by introducing an efficient cross-shard commit protocol that atomically handles transactions affecting multiple shards. OmniLedger also optimizes performance via parallel intra-shard transaction processing, ledger pruning via collectively-signed state blocks, and low-latency "trust-but-verify" validation for low-value transactions. An evaluation of our experimental prototype shows that OmniLedger's throughput scales linearly in the number of active validators, supporting Visa-level workloads and beyond, while confirming typical transactions in under two seconds.**

I. Introduction

The scalability of distributed ledgers (DLs), in both total transaction volume and the number of independent participants involved in processing them, is a major challenge to their mainstream adoption, especially when weighted against security and decentralization challenges. Many approaches exhibit different security and performance trade-offs [10], [11], [21], [32], [40]. Replacing the Nakamoto consensus [36] with PBFT [13], for example, can increase throughput while decreasing transaction commit latency [1], [32]. These approaches still require all *validators* or consensus group members to redundantly validate and process all transactions, hence the system's total transaction processing capacity does not increase with added participants, and, in fact, gradually decreases due to increased coordination overheads.

The proven and obvious approach to building "*scale-out*" databases, whose capacity scales horizontally with the number of participants, is by *sharding* [14], or partitioning the state into multiple shards that are handled in parallel by different subsets of participating validators. Sharding could benefit DLs [15] by reducing the transaction processing load on each validator and by increasing the system's total processing capacity proportionally with the number of participants. Existing proposals for sharded DLs, however, forfeit permissionless decentralization [16], introduce new security assumptions, and/or trade performance for security [34], as illustrated in Figure 1 and explored in detail in Sections II and IX.

We introduce OmniLedger, the first DL architecture that provides "scale-out" transaction processing capacity competitive with centralized payment-processing systems, such as Visa, without compromising security or support for permissionless decentralization. To achieve this goal, OmniLedger faces three key correctness and security challenges. First, OmniLedger must choose statistically representative groups of validators periodically via permissionless Sybil-attack-resistant foundations such as proof-of-work [36], [38], [32] or proof-of-stake [31], [25]. Second, OmniLedger must ensure a negligible probability that any shard is compromised across the (long-term) system lifetime via periodically (re-)forming shards (subsets of validators to record state and process transactions), that are both sufficiently large and bias-resistant. Third, OmniLedger must correctly and atomically handle *cross-shard transactions*, or transactions that affect the ledger state held by two or more distinct shards.

Fig. 1: Trade-offs in current DL systems.

To choose representative validators via proof-of-work, OmniLedger builds on ByzCoin [32] and Hybrid Consensus [38], using a sliding window of recent proof-of-work block miners as its validator set. To support the more power-efficient alternative of apportioning consensus group membership based on directly invested stake rather than work, OmniLedger builds on Ouroboros [31] and Algorand [25], running a public randomness or cryptographic sortition protocol within a prior validator group to pick a subsequent validator group from the current stakeholder distribution defined in the ledger. To ensure that this sampling of representative validators is both scalable and strongly bias-resistant, OmniLedger uses RandHound [44], a protocol that serves this purpose under standard t-of-n threshold assumptions.

Appropriate use of RandHound provides the basis by which OmniLedger addresses the second key security challenge of securely assigning validators to shards, and of periodically rotating these assignments as the set of validators evolves. OmniLedger chooses shards large enough, based on the analysis in Section VI, to ensure a negligible probability that any shard is ever compromised, even across years of operation.

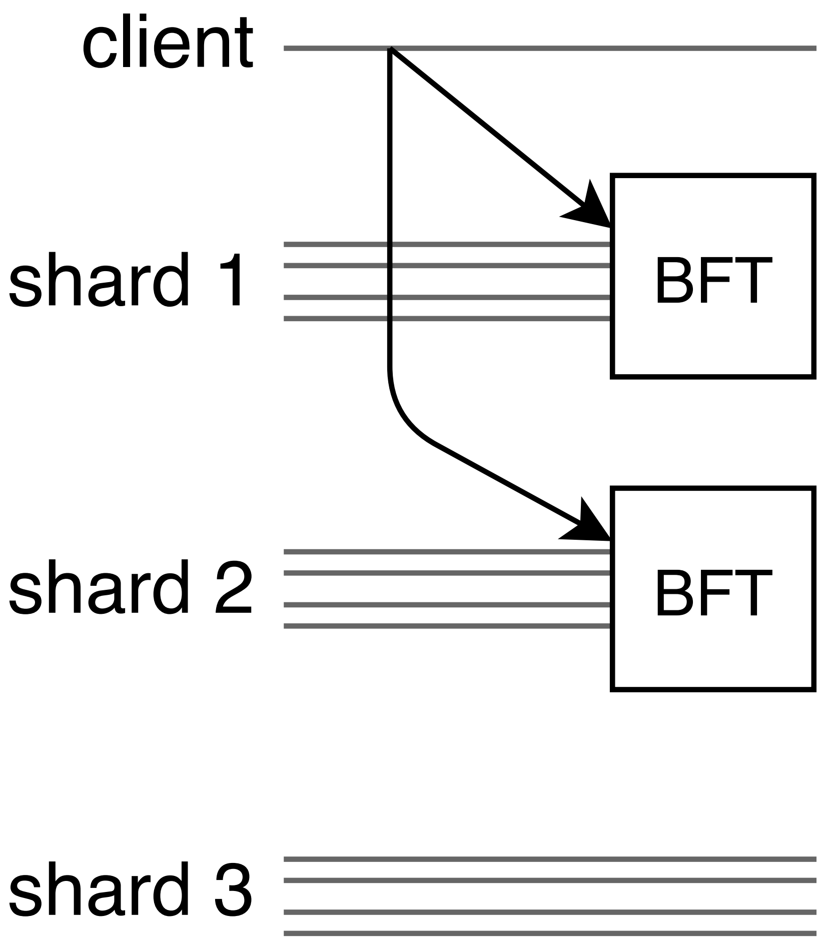
Finally, to ensure that transactions either commit or abort atomically even when they affect state distributed across multiple shards (*e.g.*, several cryptocurrency accounts), OmniLedger introduces Atomix, a two-phase client-driven "lock/unlock" protocol that ensures that clients can either fully commit a transaction across shards, or obtain "rejection proofs" to abort and unlock state affected by partially completed transactions.

S&P'18

Shard-Led Cross-Shard Consensus

- Chainspace

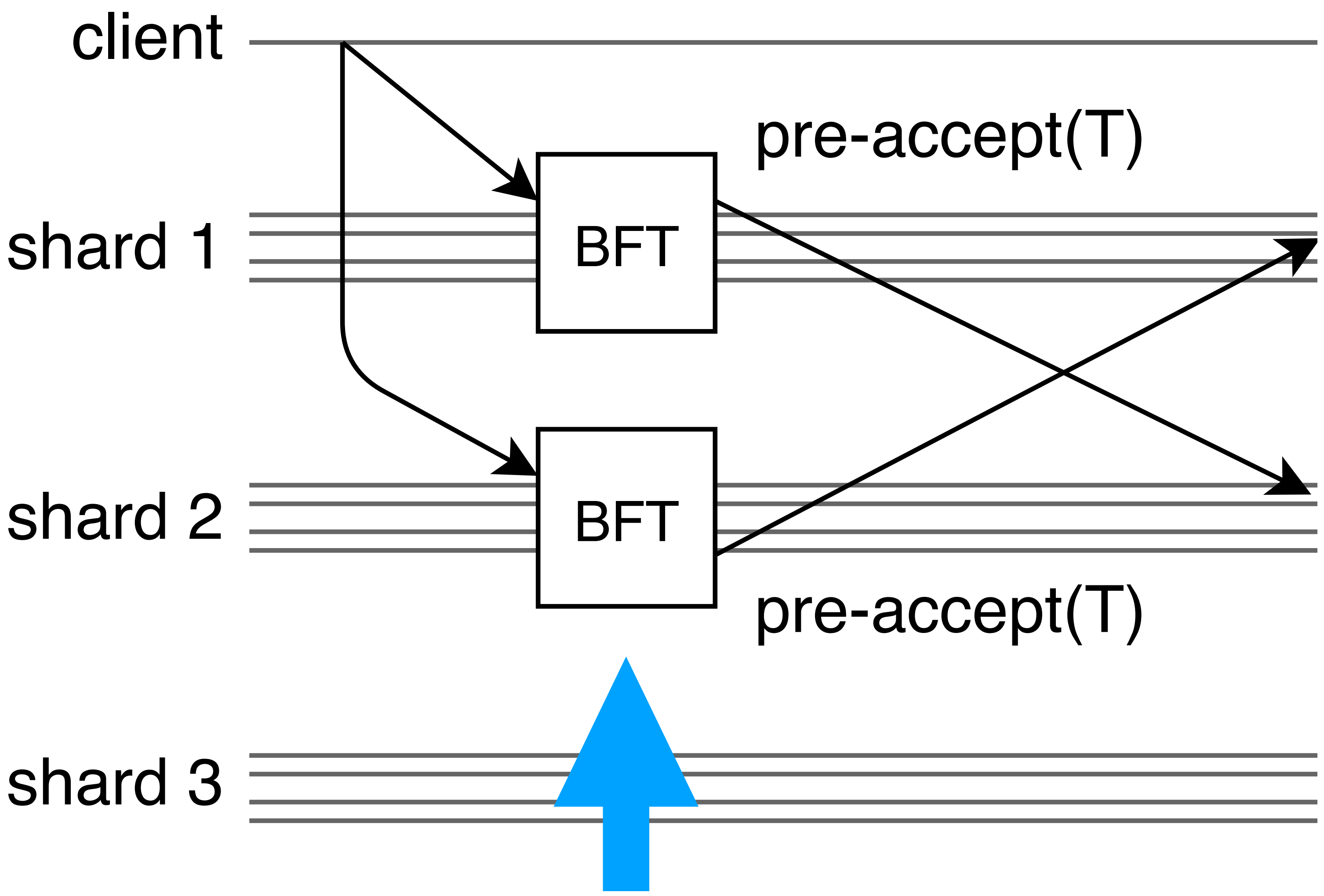
$$T(x_1, x_2) \rightarrow (y_1, y_2, y_3)$$



Shard-Led Cross-Shard Consensus

- Chainspace

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

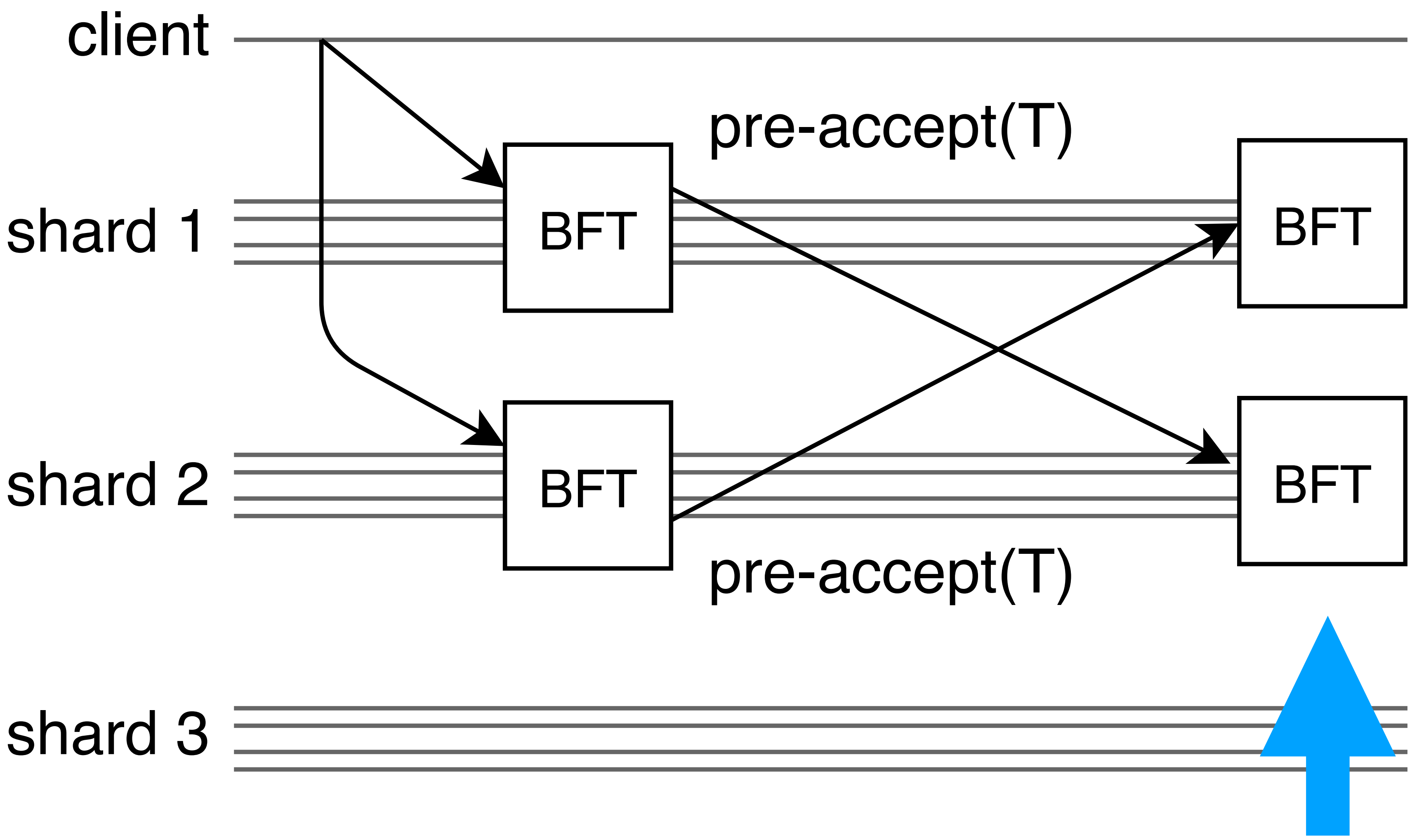


lock X_1, X_2

Shard-Led Cross-Shard Consensus

- Chainspace

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

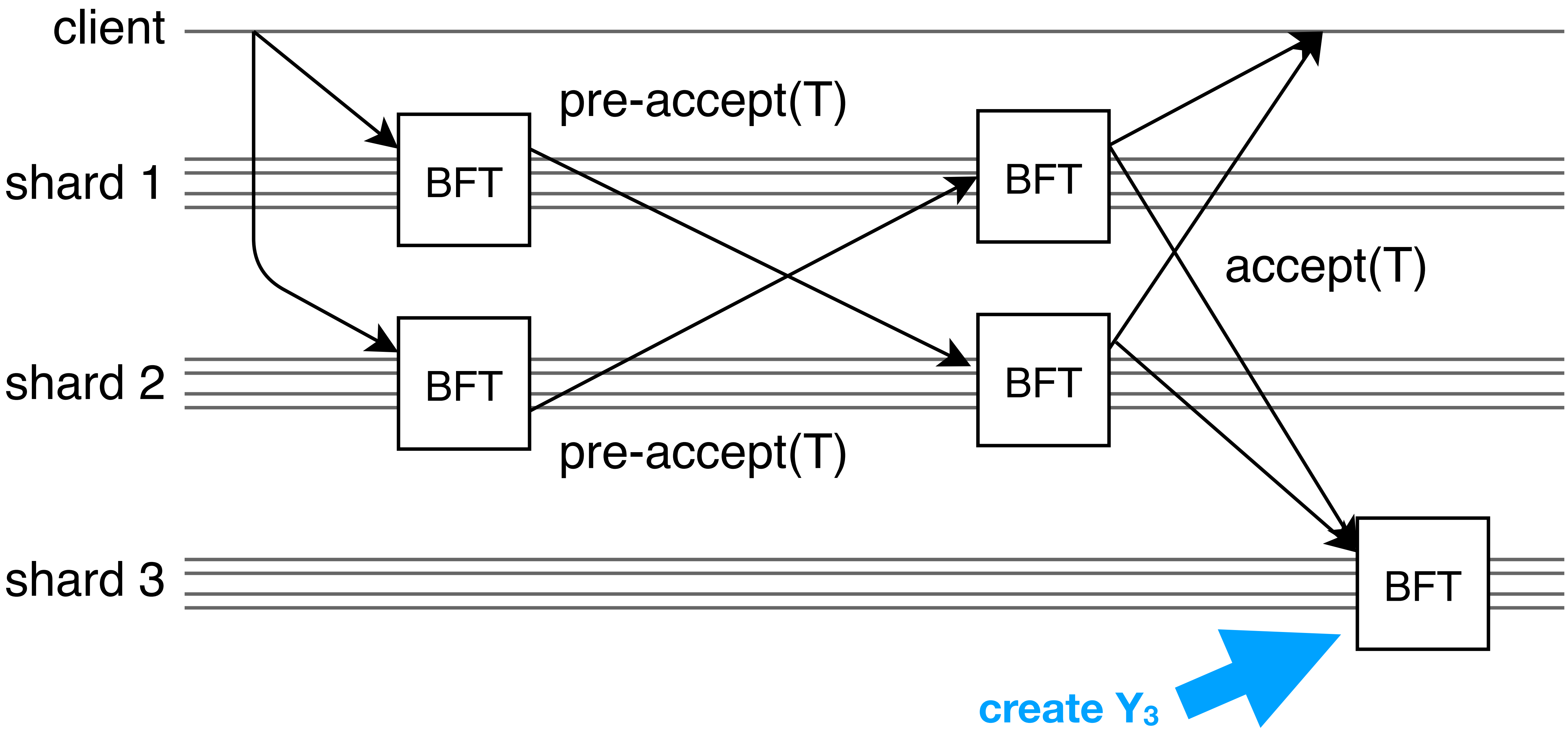


delete X_1, X_2 ; create Y_1, Y_2

Shard-Led Cross-Shard Consensus

- Chainspace

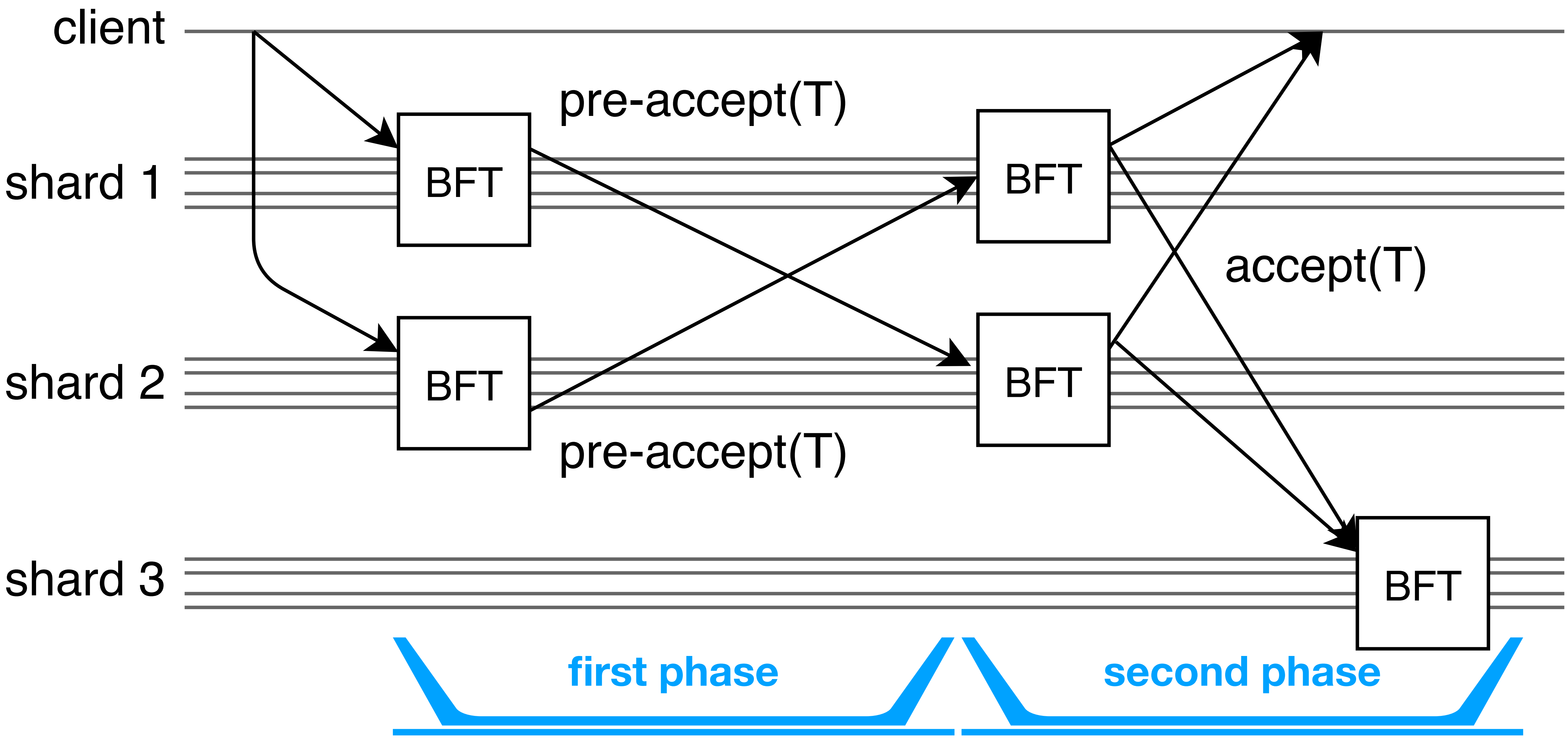
$$T(x_1, x_2) \rightarrow (y_1, y_2, y_3)$$



Shard-Led Cross-Shard Consensus

- Chainspace

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



Shard-Led Cross-Shard Consensus

- First phase attacks

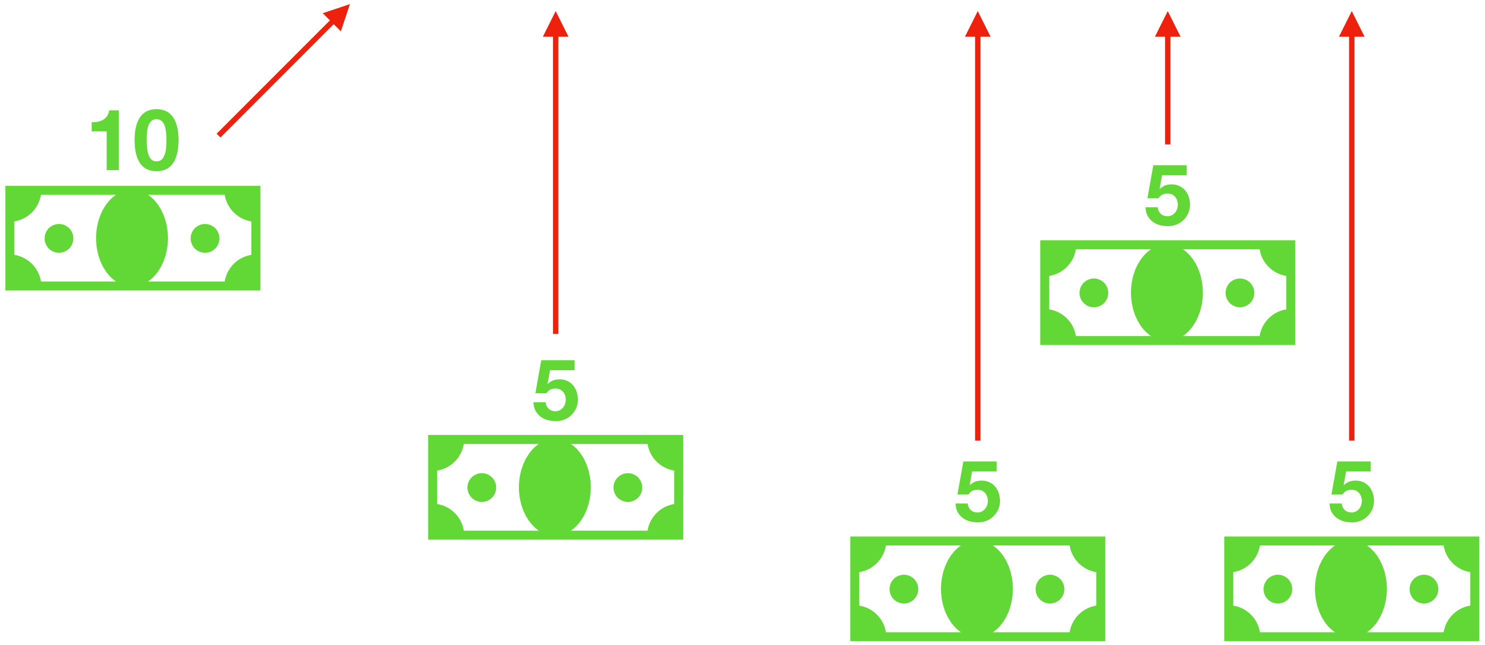
	Phase 1 of S-BAC		Phase 2 of S-BAC		
	Shard 1 (potential victim)	**Shard 2** (potential victim)	**Shard 1** (potential victim)	**Shard 2** (potential victim)	**Shard 3** (potential victim)
1	pre-accept(T) lock x_1	pre-accept(T) lock x_2	accept(T) create y_1; inactivate x_1	accept(T) create y_2; inactivate 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) -	-

Shard-Led Cross-Shard Consensus

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

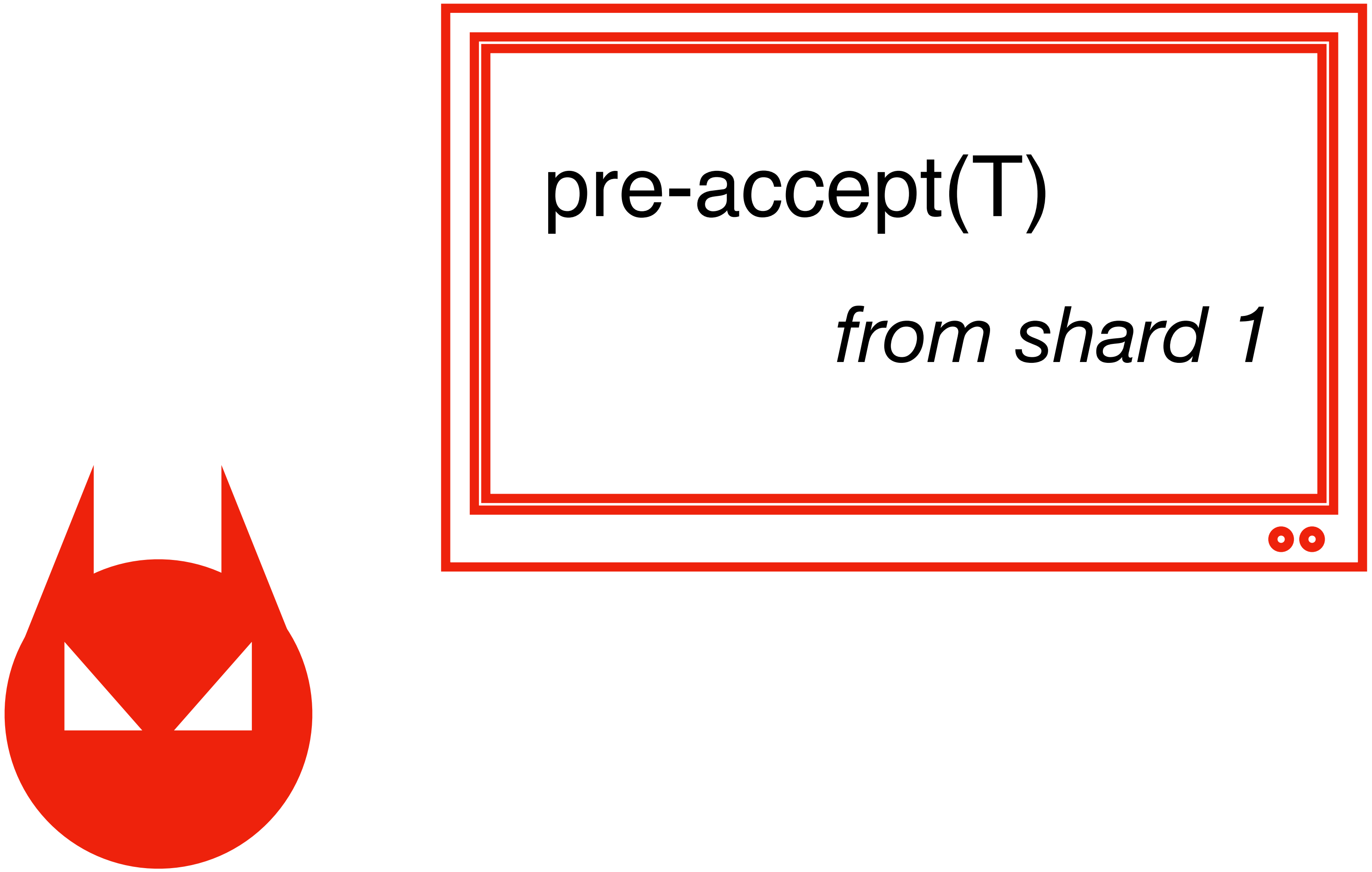
transaction

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



Shard-Led Cross-Shard Consensus

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



pre-accept(T)

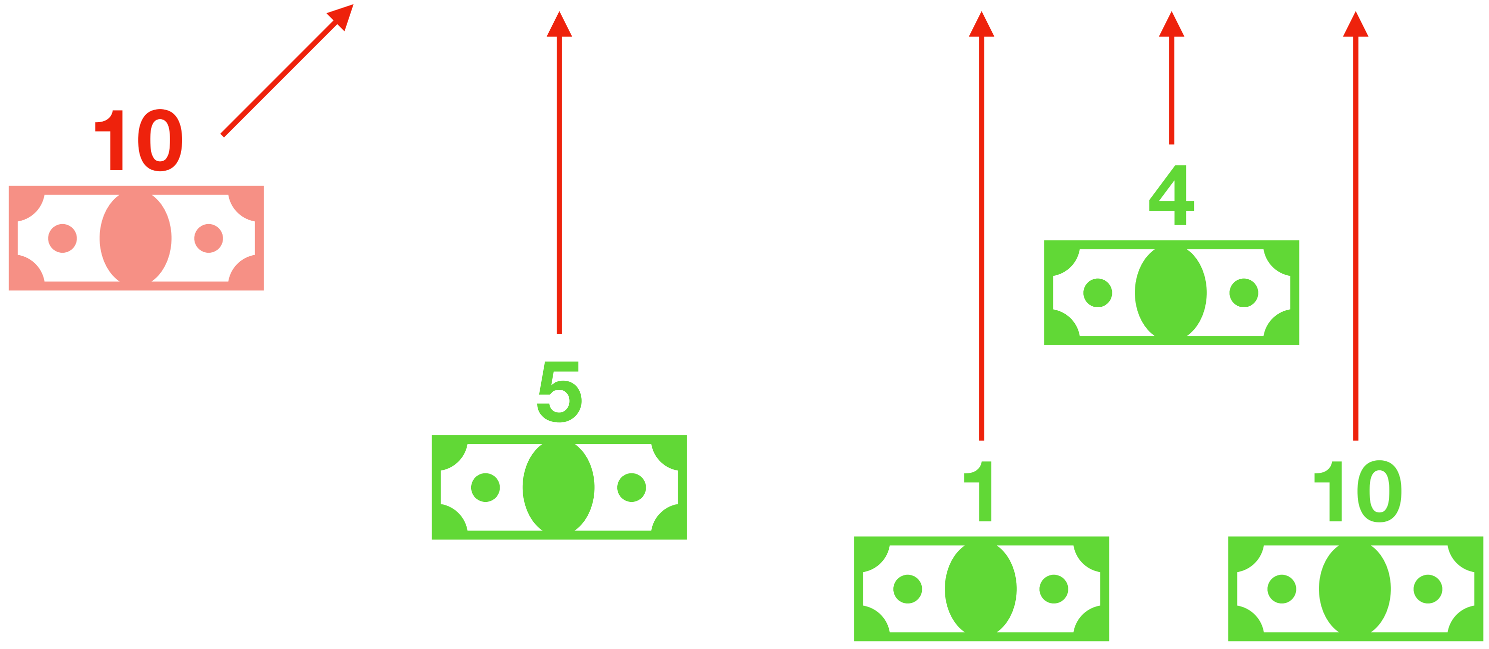
from shard 1

Shard-Led Cross-Shard Consensus

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

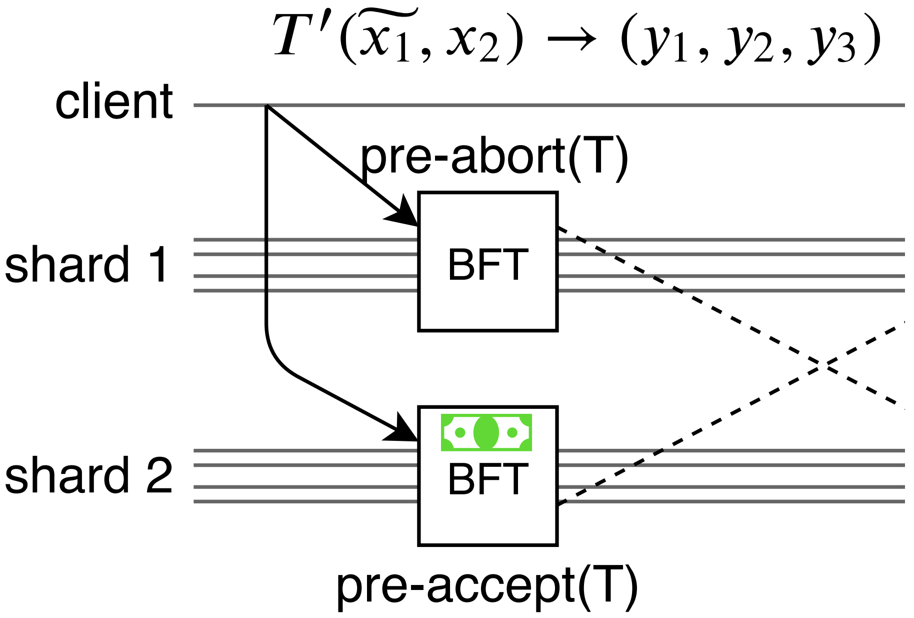
(bad) transaction

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



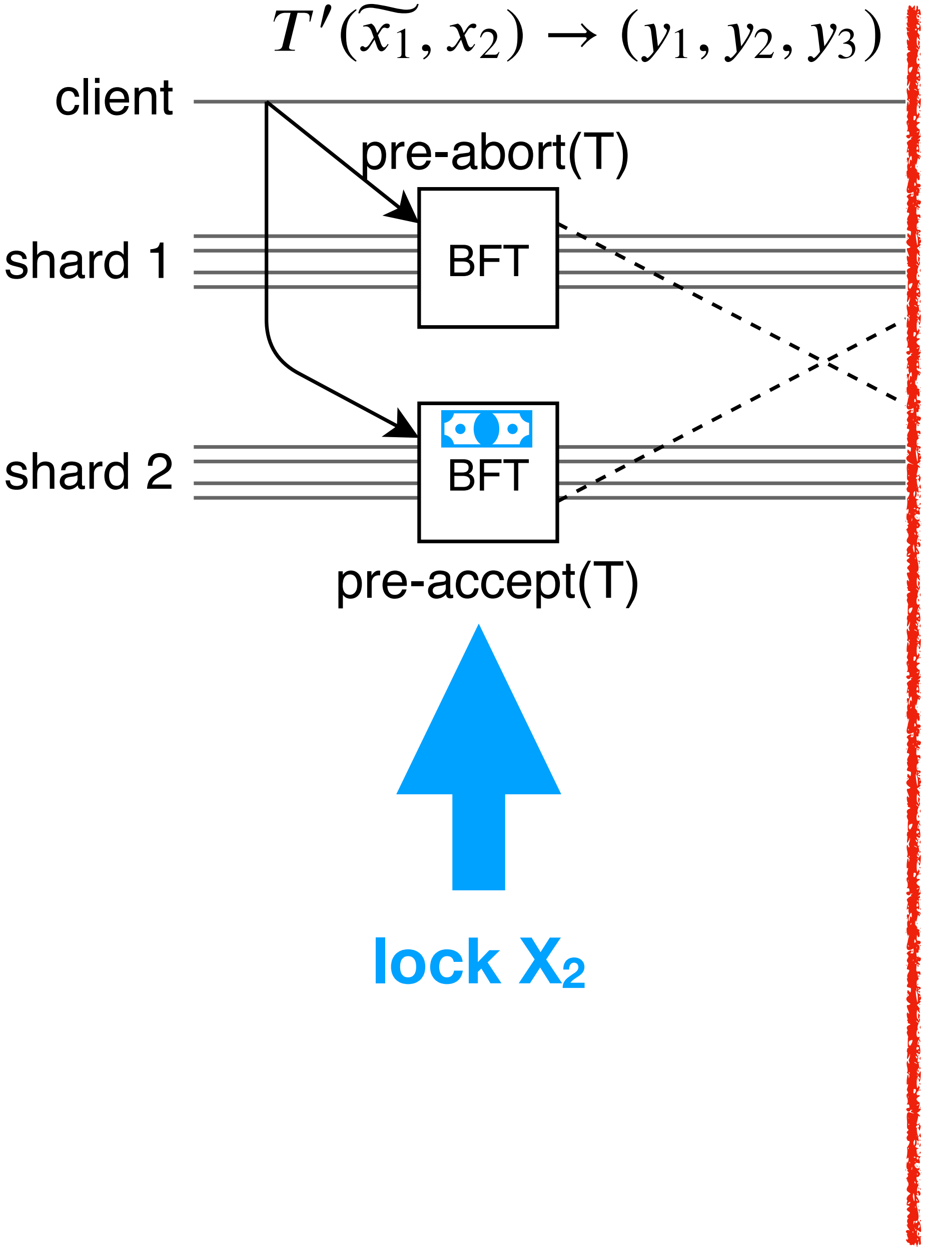
Shard-Led Cross-Shard Consensus

- First phase attacks: recording messages



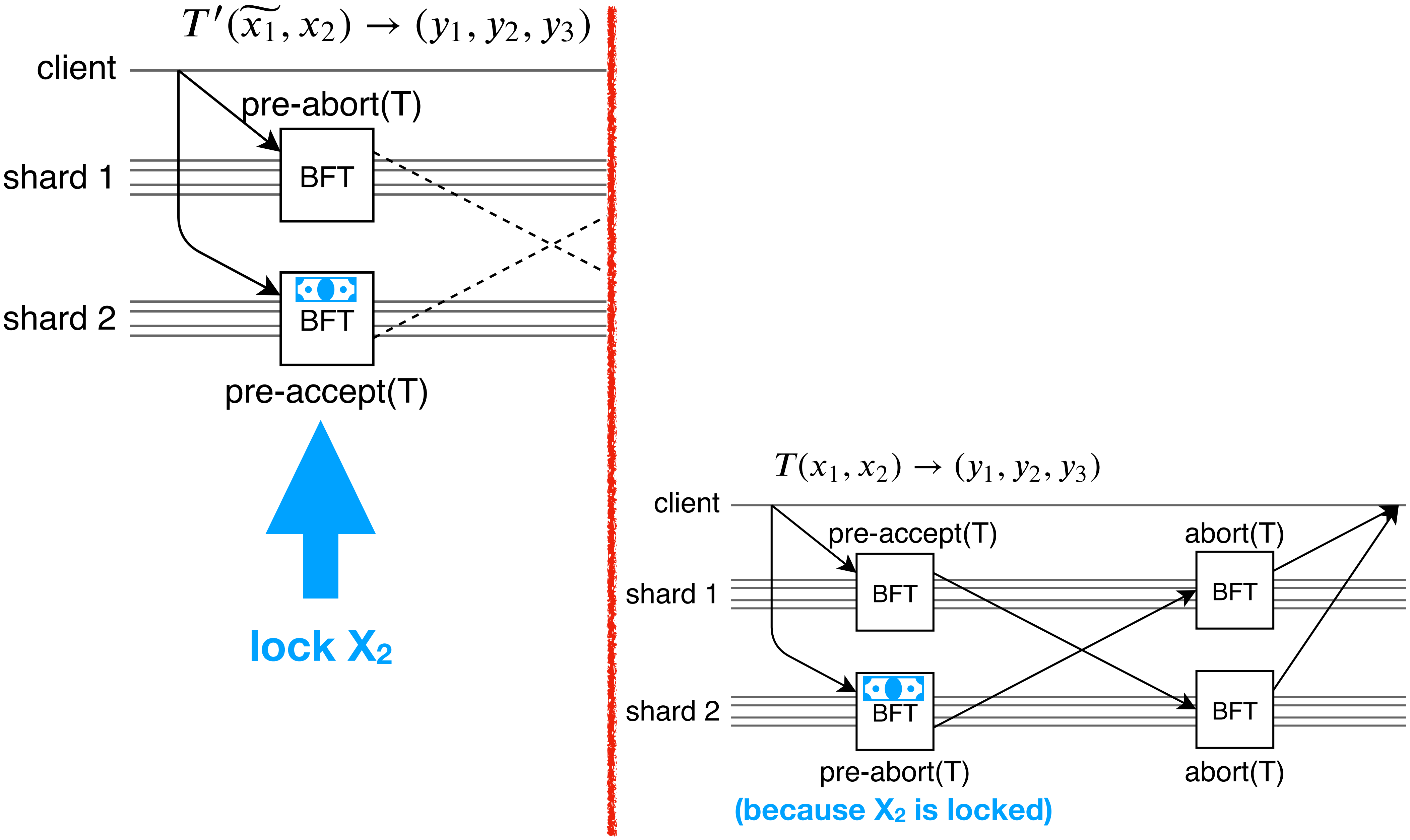
Shard-Led Cross-Shard Consensus

- First phase attacks: recording messages



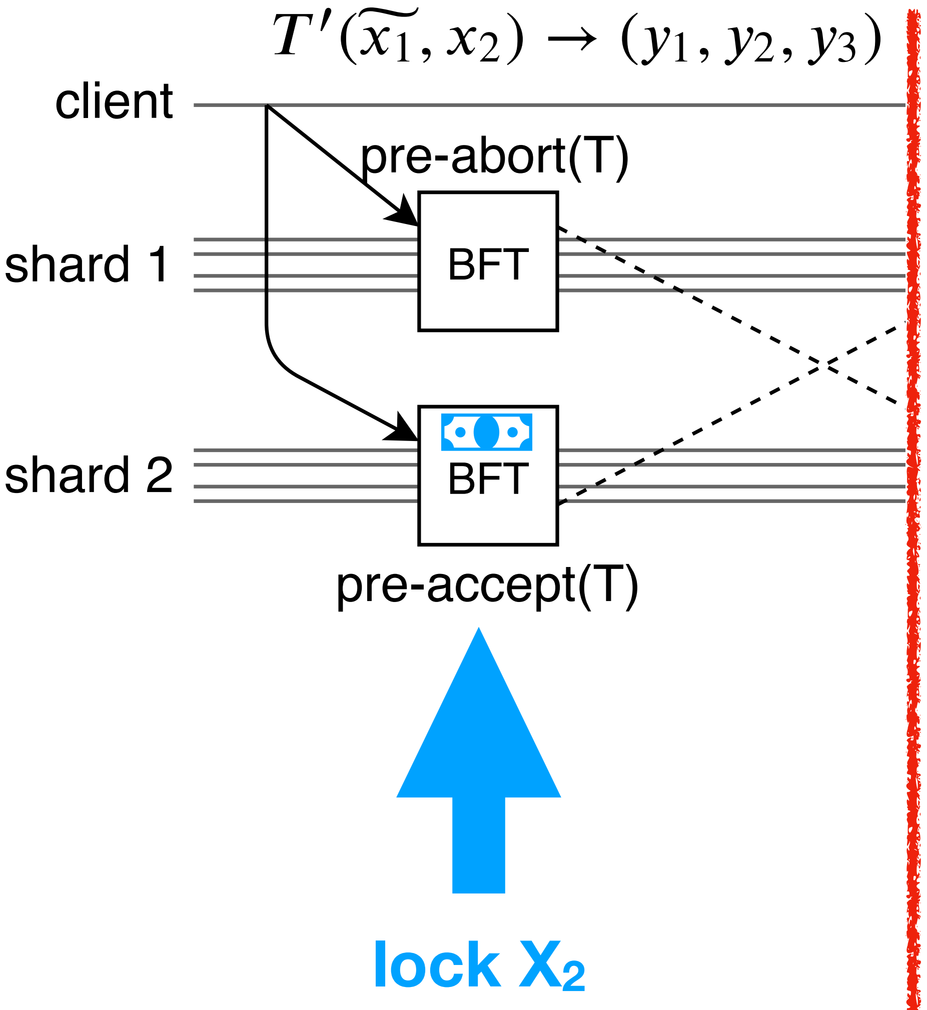
Shard-Led Cross-Shard Consensus

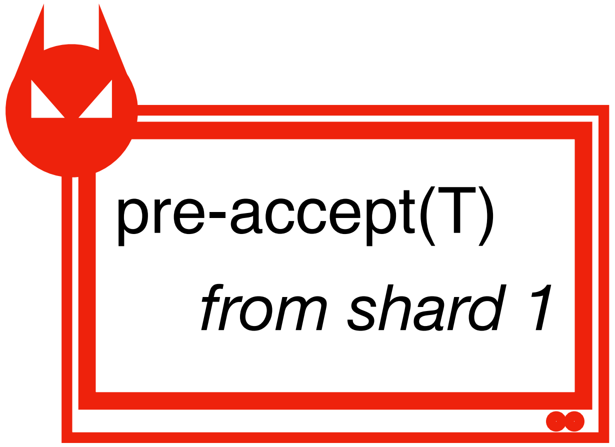
- First phase attacks: recording messages

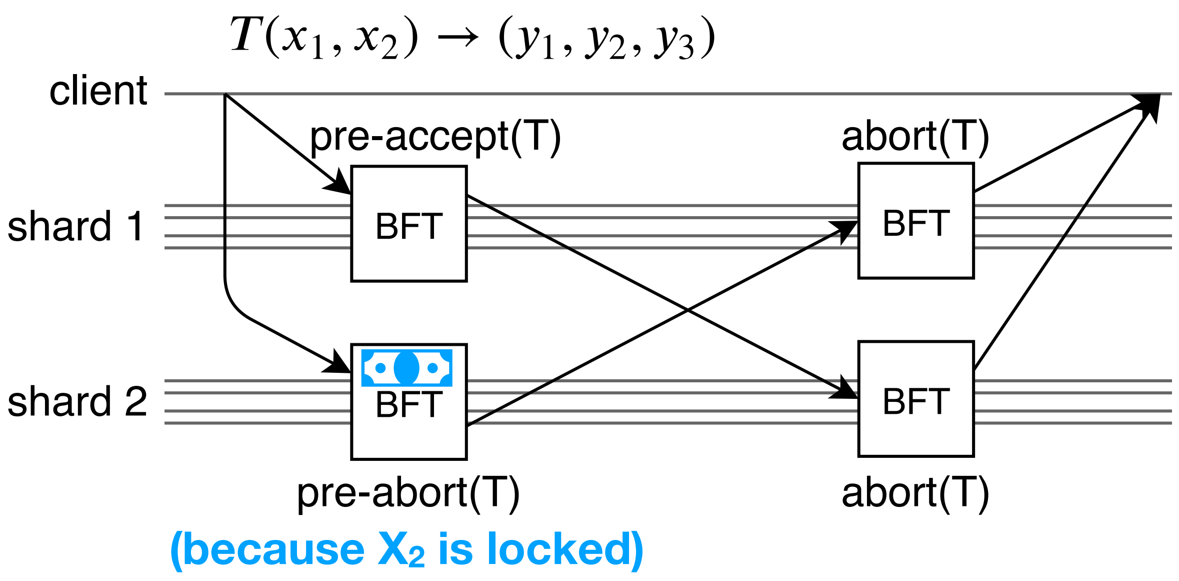


Shard-Led Cross-Shard Consensus

- First phase attacks: recording messages

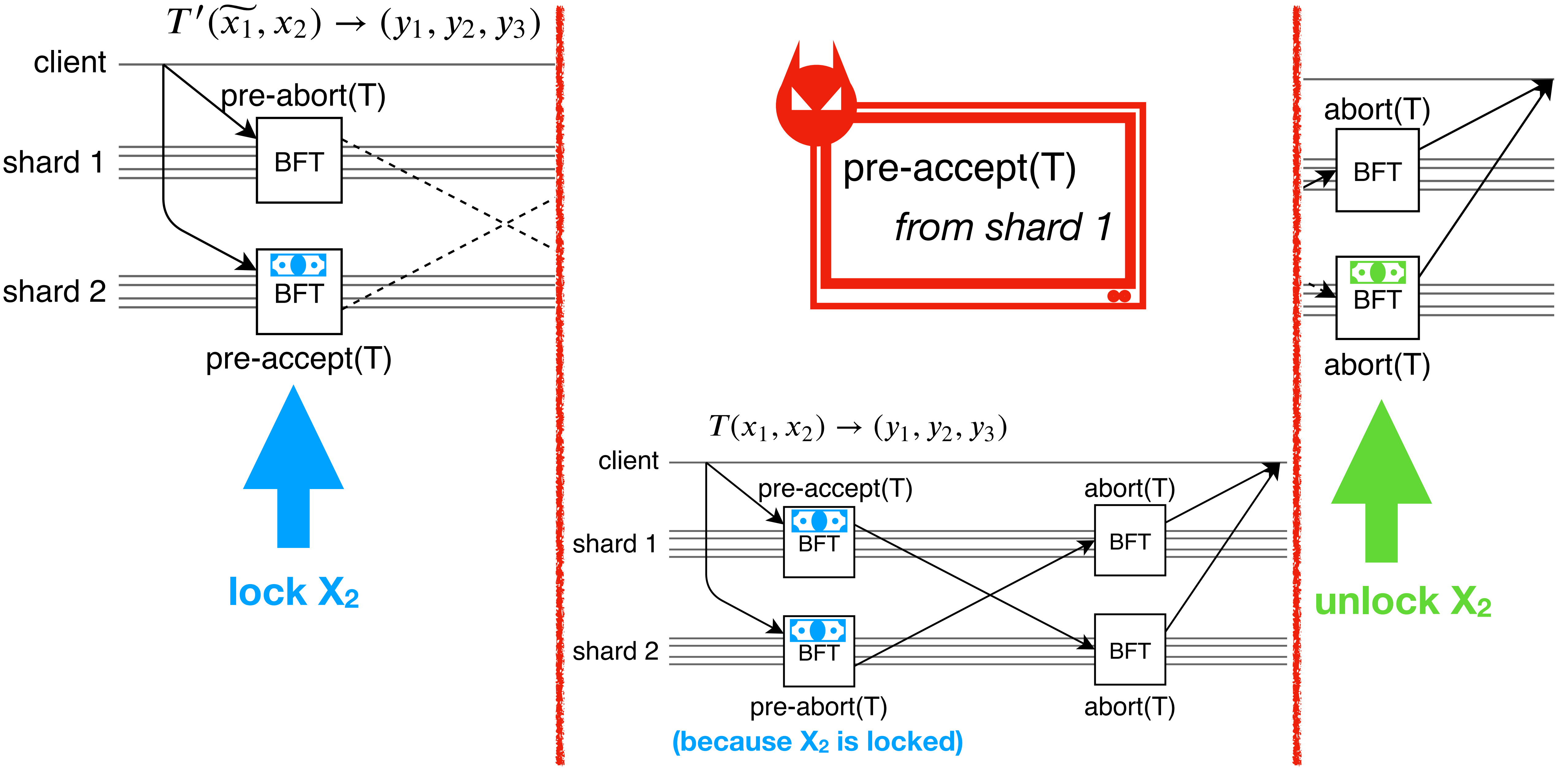






Shard-Led Cross-Shard Consensus

- First phase attacks: recording messages

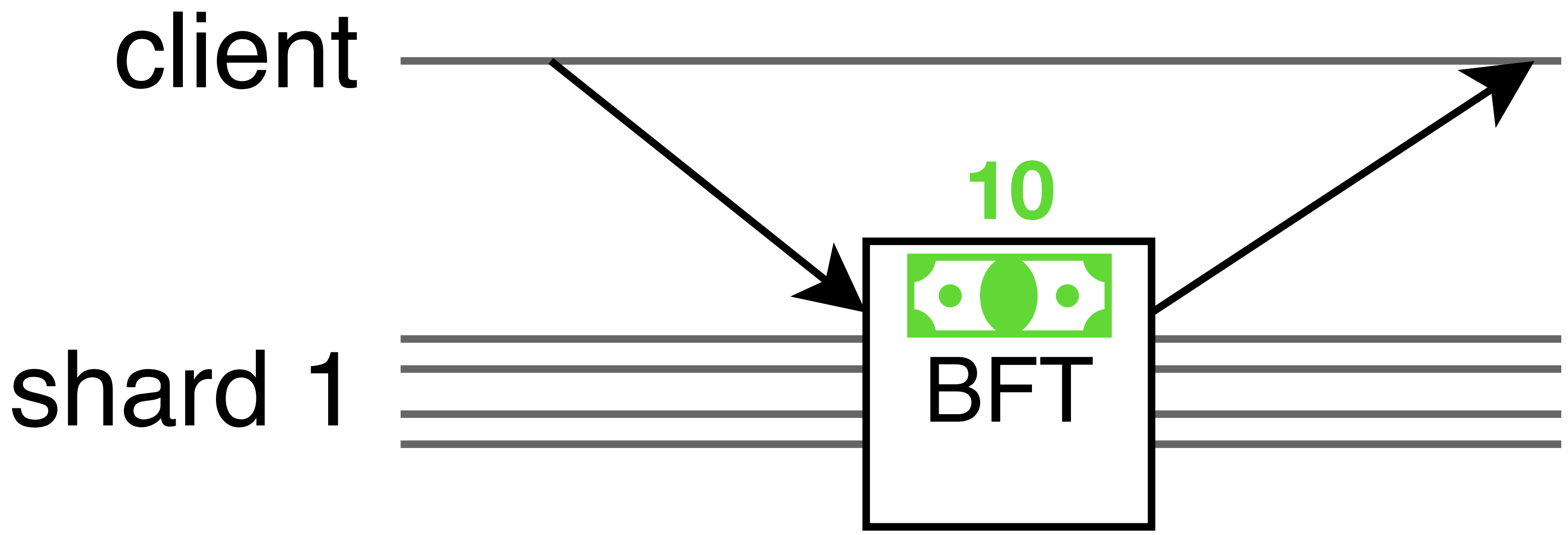


Shard-Led Cross-Shard Consensus

pre-accept(T)
from shard 1

- First phase attacks: spend X_1

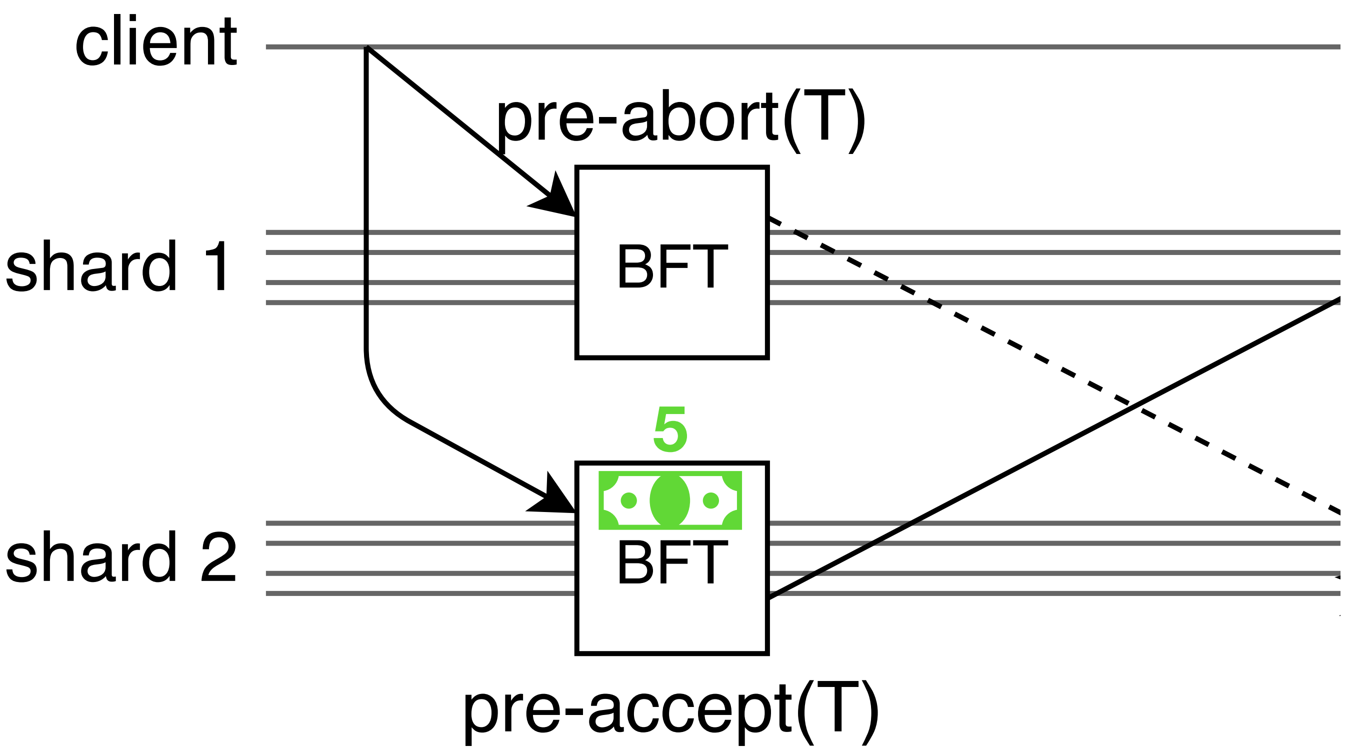
$$T^*(x_1) \rightarrow (y_*)$$

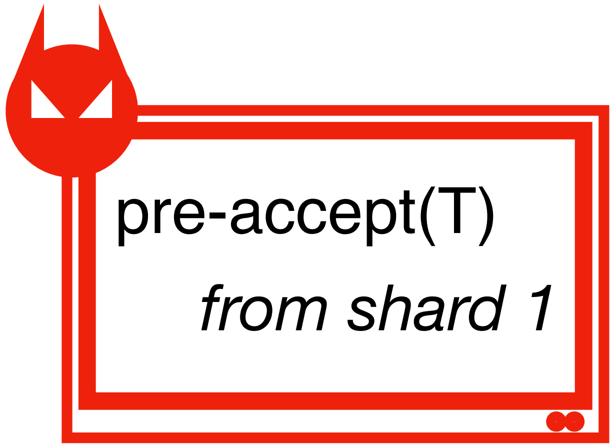


Shard-Led Cross-Shard Consensus

- First phase attacks: double-spend X_1

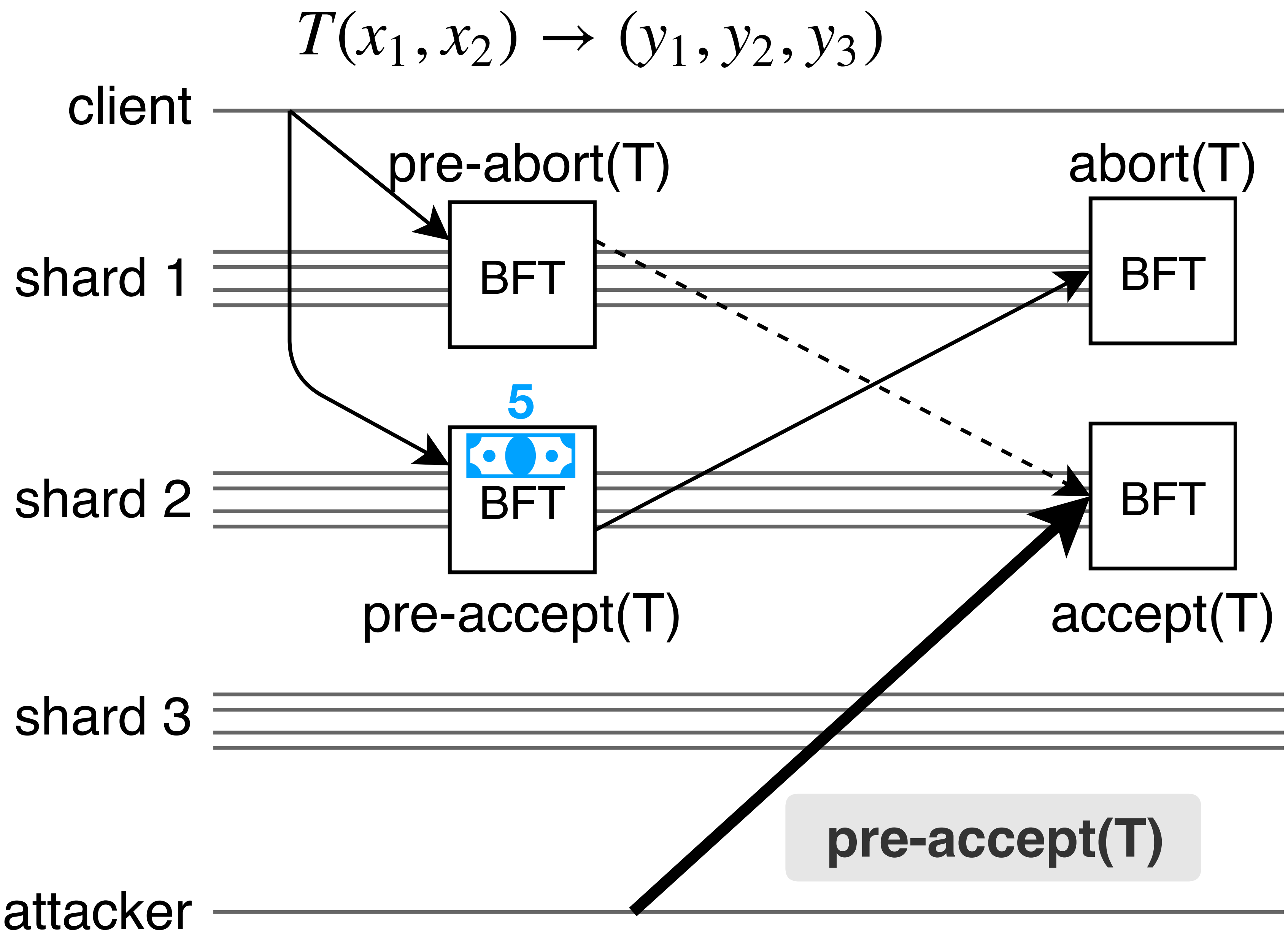
$$T(x_1, x_2) \rightarrow (y_1, y_2, y_3)$$

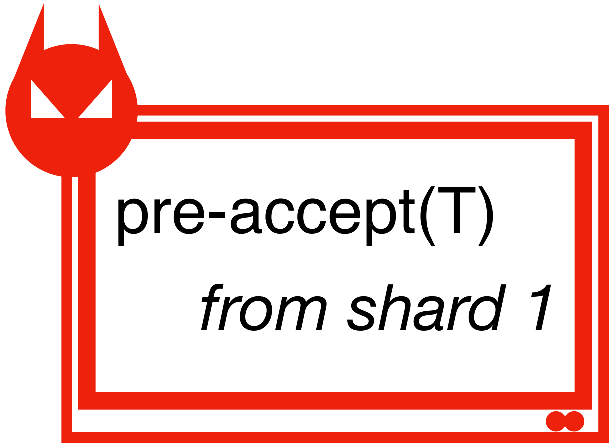




Shard-Led Cross-Shard Consensus

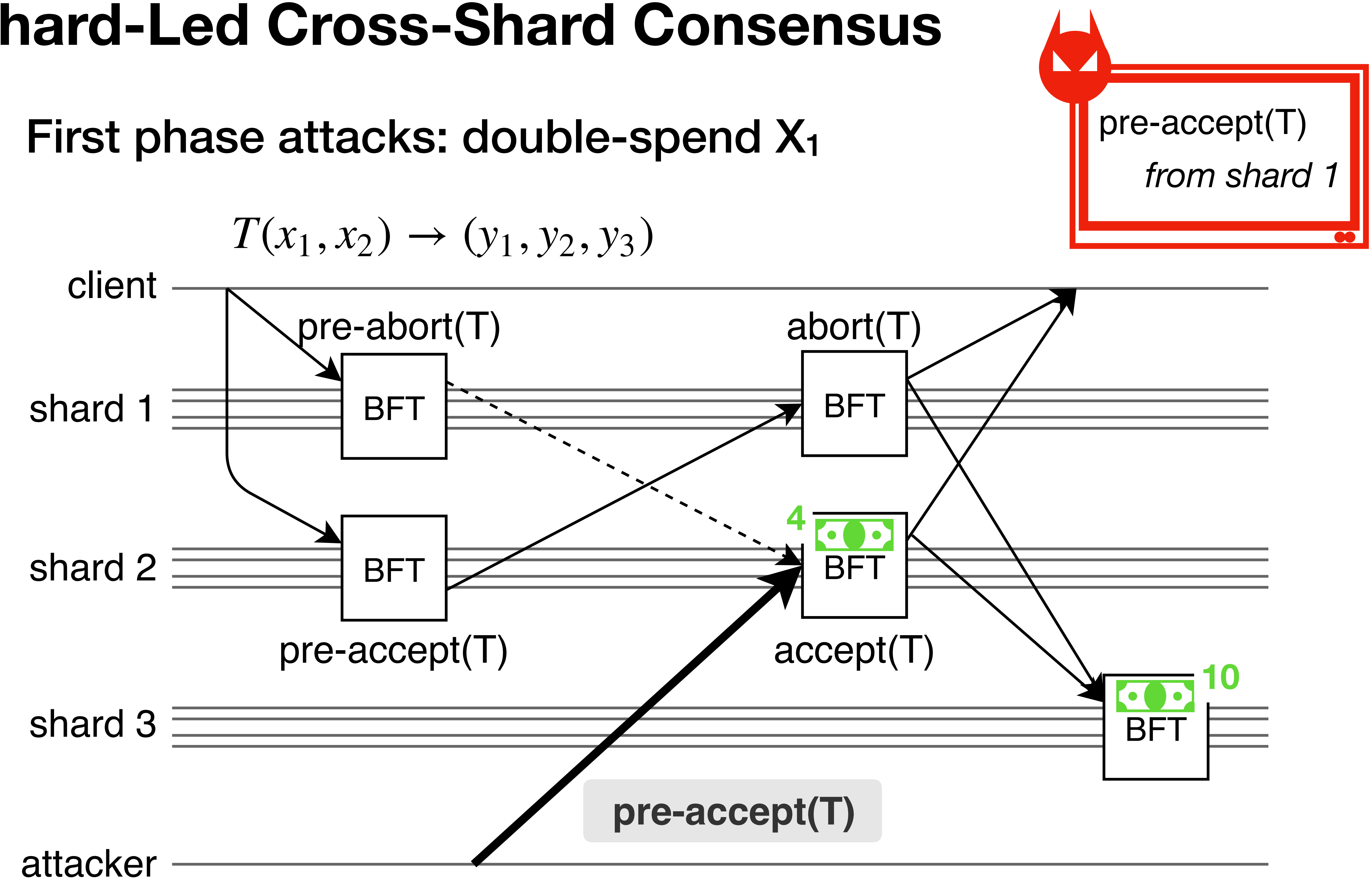
- First phase attacks: double-spend X_1





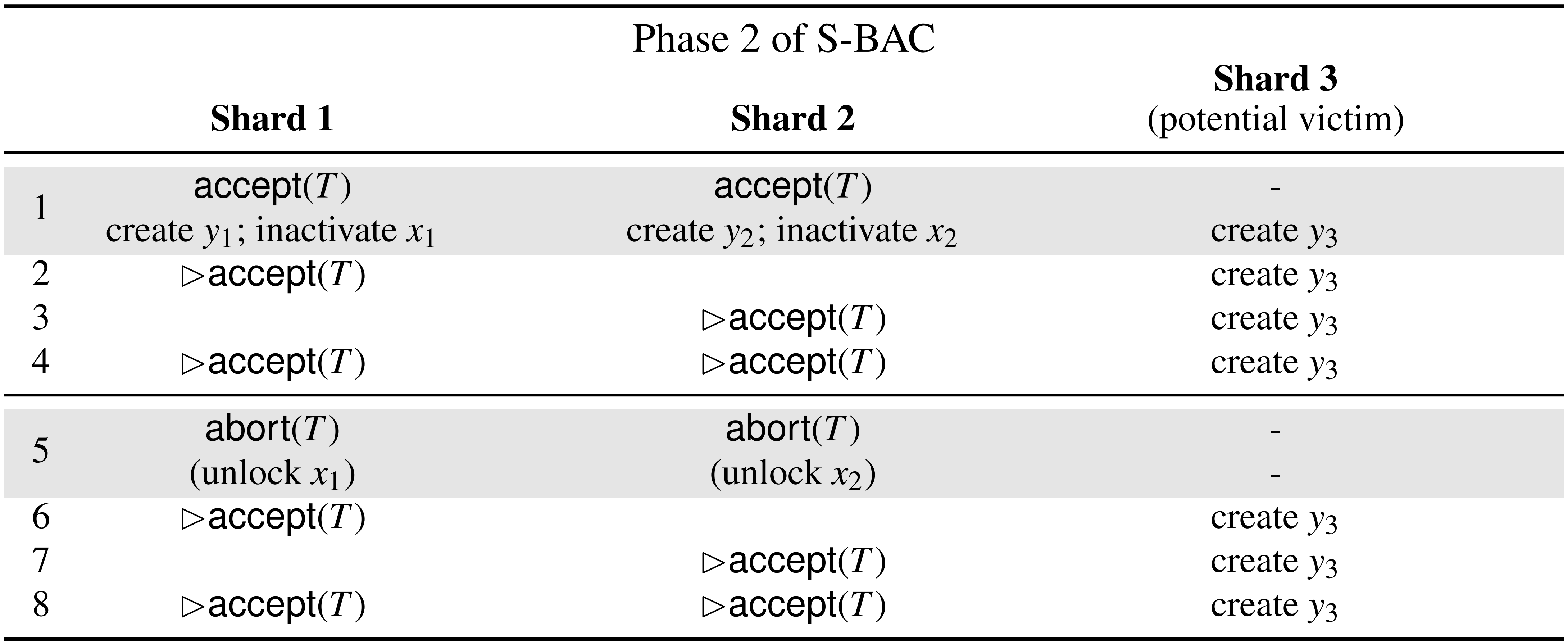
Shard-Led Cross-Shard Consensus

- **First phase attacks: double-spend X_1**



Shard-Led Cross-Shard Consensus

- Second phase



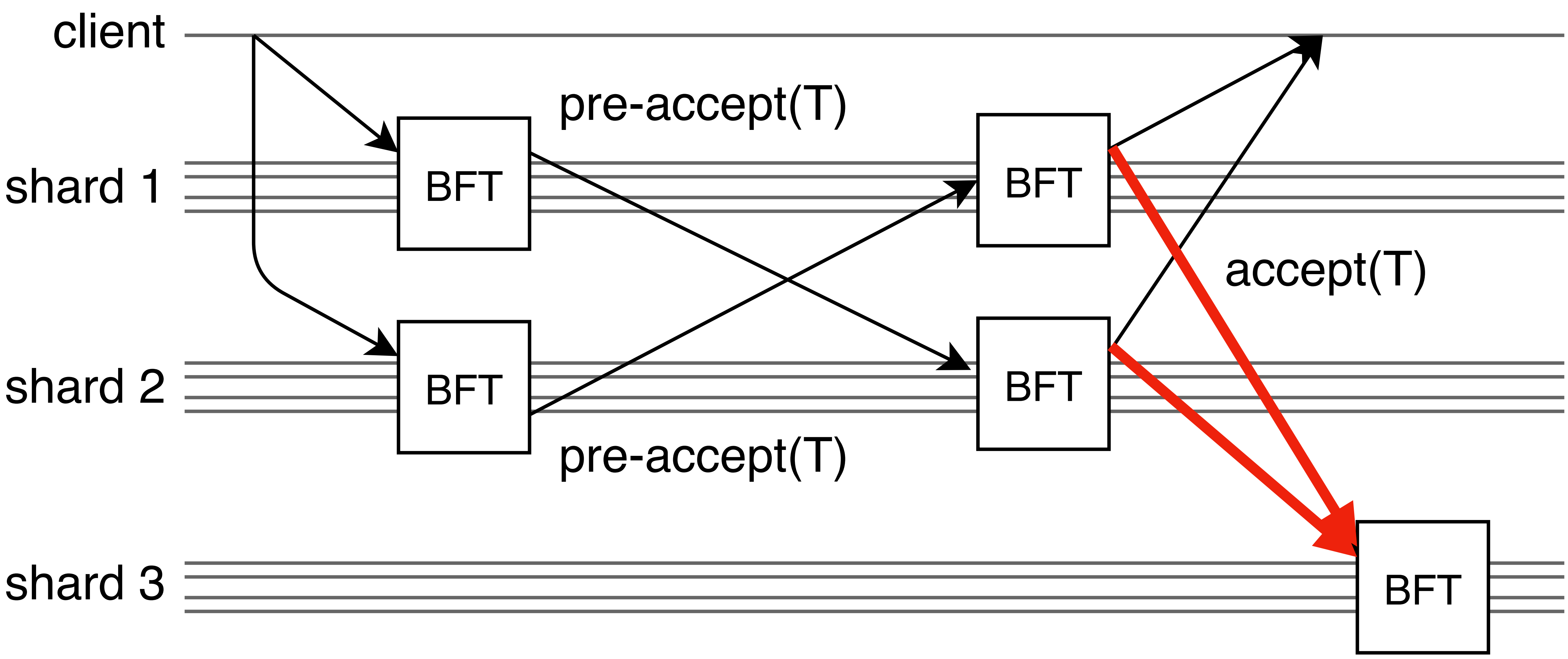
Phase 2 of S-BAC

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

Shard-Led Cross-Shard Consensus

- Second phase

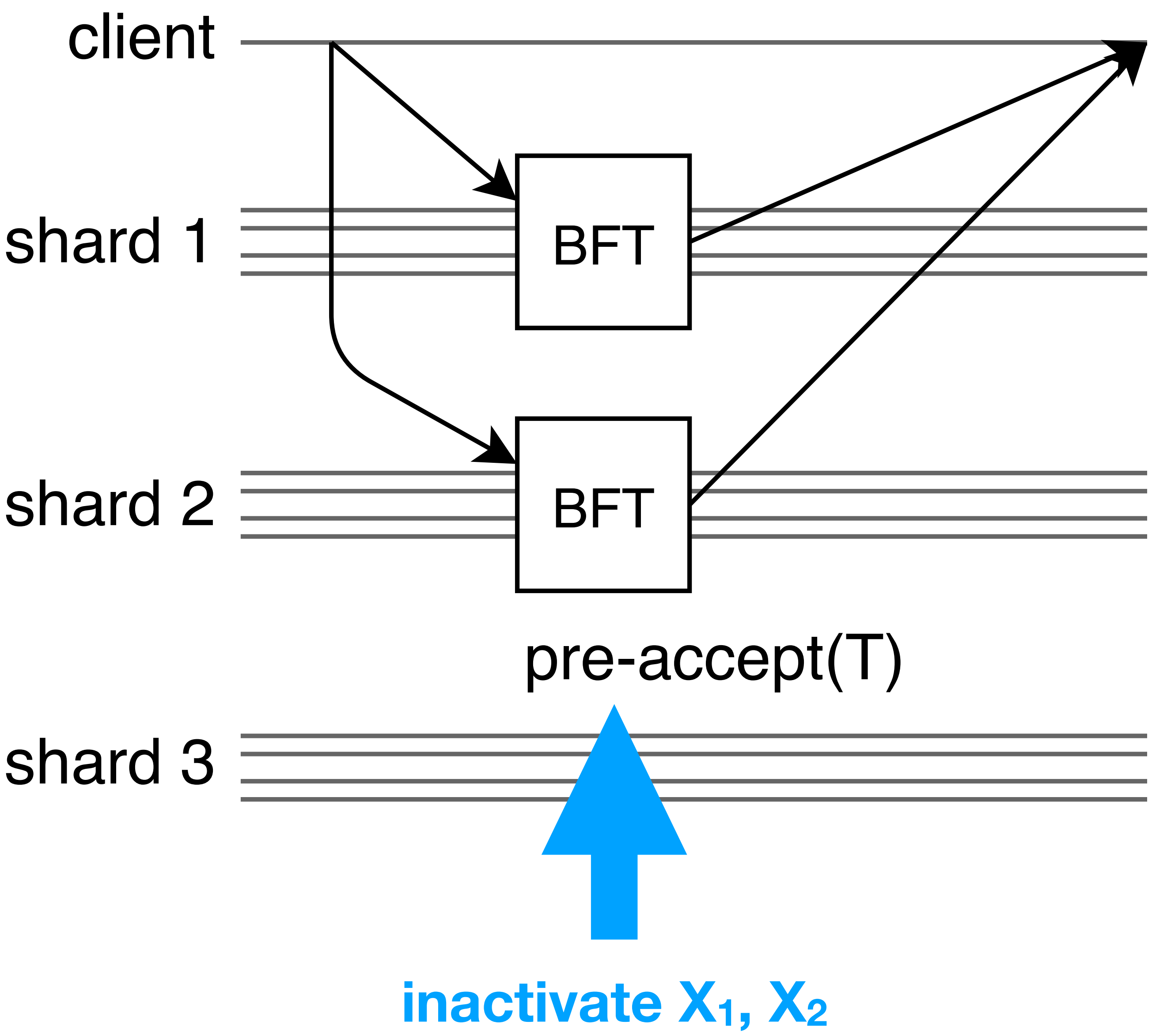
$T(x_1, x_2) \rightarrow (y_1, y_2, y_3)$



Client-Led Cross-Shard Consensus

- Omniledger

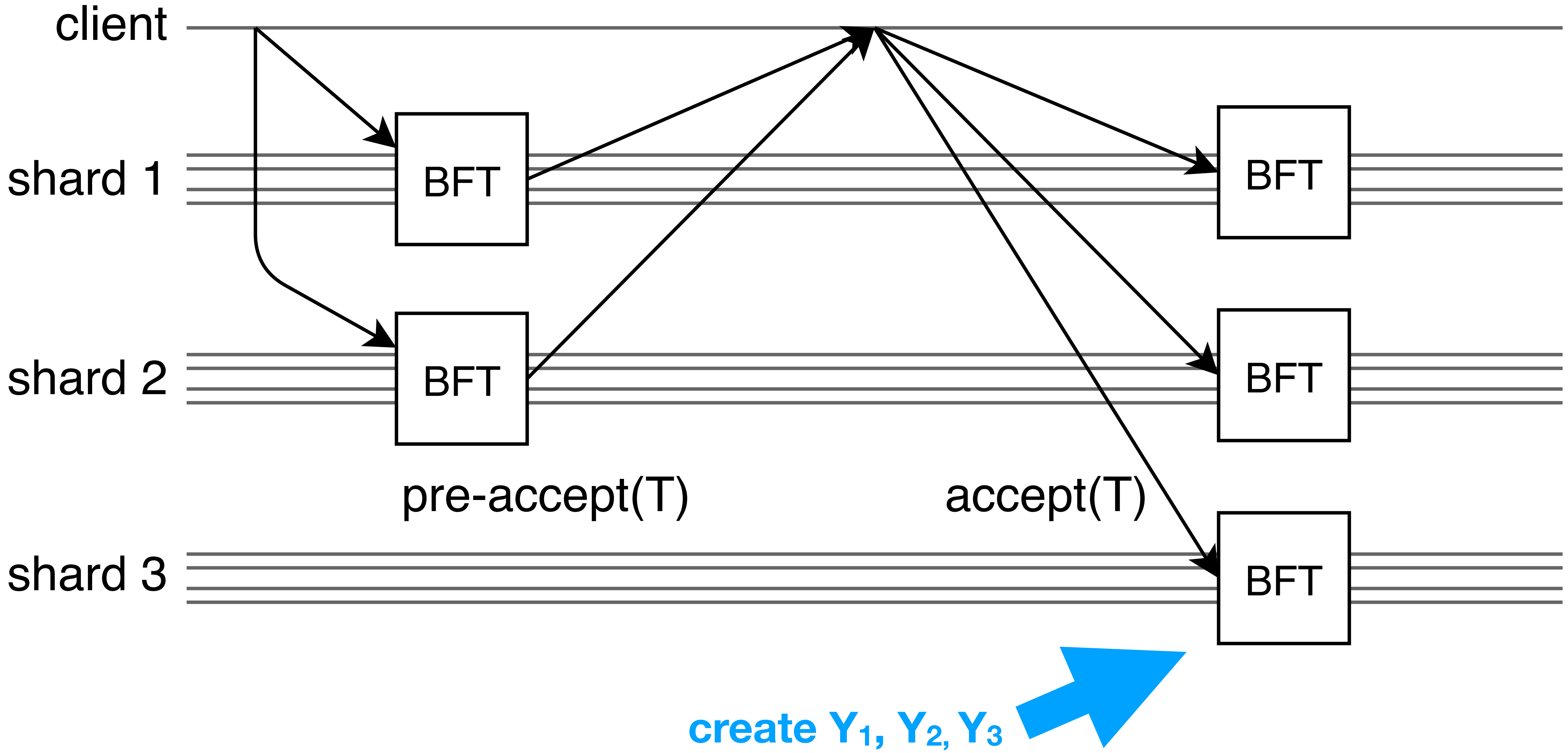
$T(x_1, x_2) \rightarrow (y_1, y_2, y_3)$



Client-Led Cross-Shard Consensus

- Omniledger

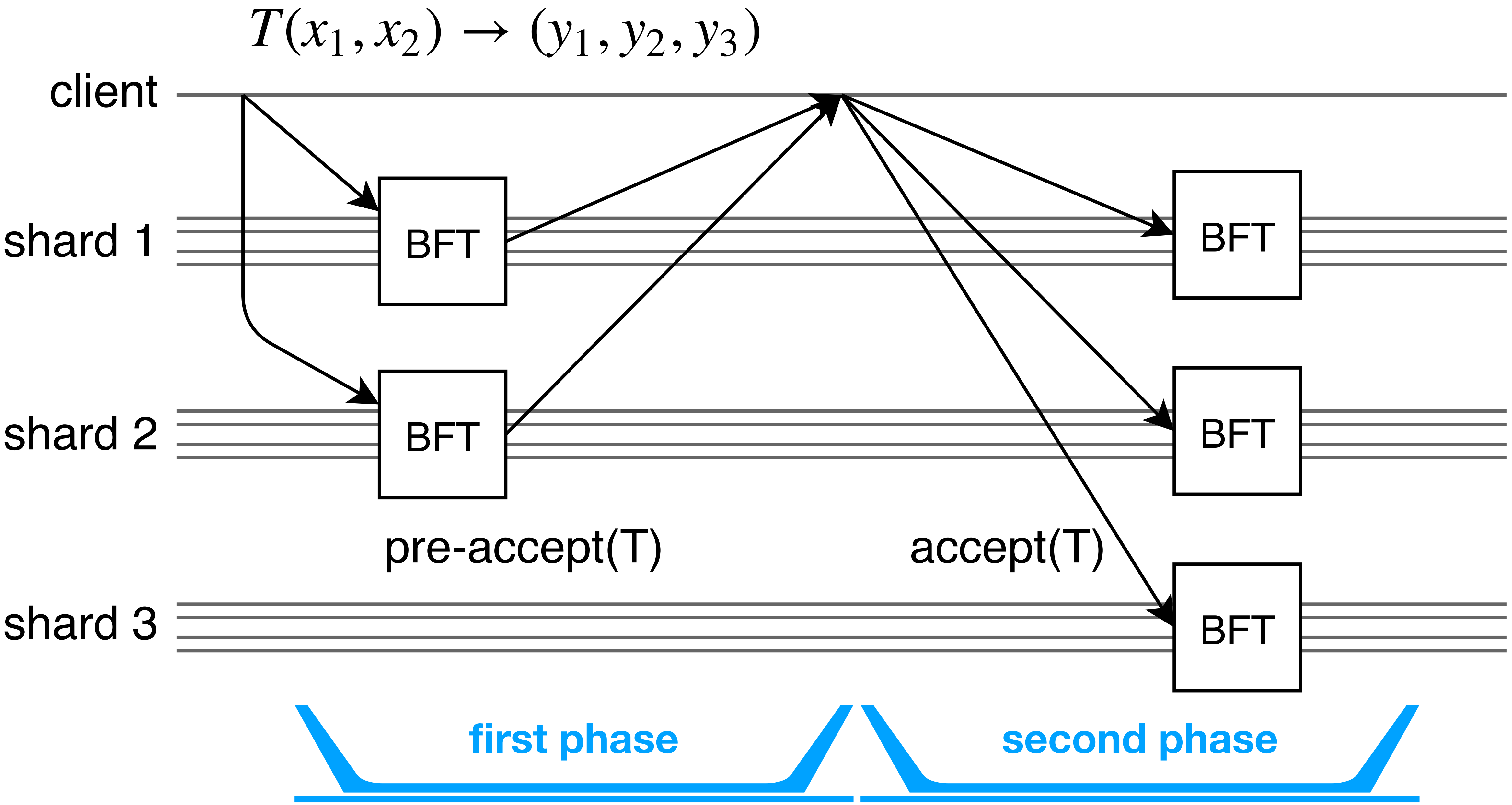
$T(x_1, x_2) \rightarrow (y_1, y_2, y_3)$



create Y_1, Y_2, Y_3

Client-Led Cross-Shard Consensus

- Omniledger



Client-Led Cross-Shard Consensus

- First phase attacks

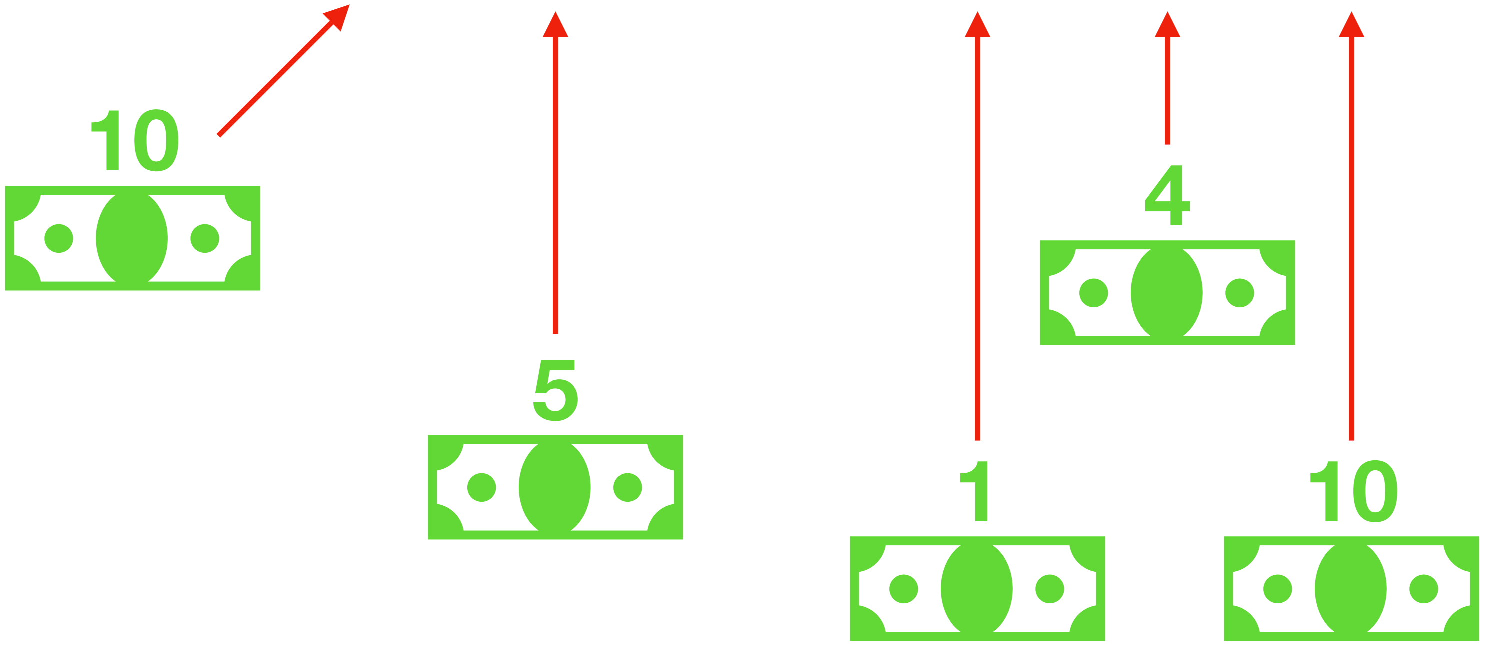
	Phase 1 of Atomix			Phase 2 of Atomix		
	Shard 1 (potential victim)	**Shard 2** (potential victim)	**Client** (victim)	**Shard 1** (potential victim)	**Shard 2** (potential victim)	**Shard 3** (potential victim)
1	pre-accept(T) inactivate x_1	pre-accept(T) inactivate x_2	accept(T)	- create y_1	- create 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

Client-Led Cross-Shard Consensus

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

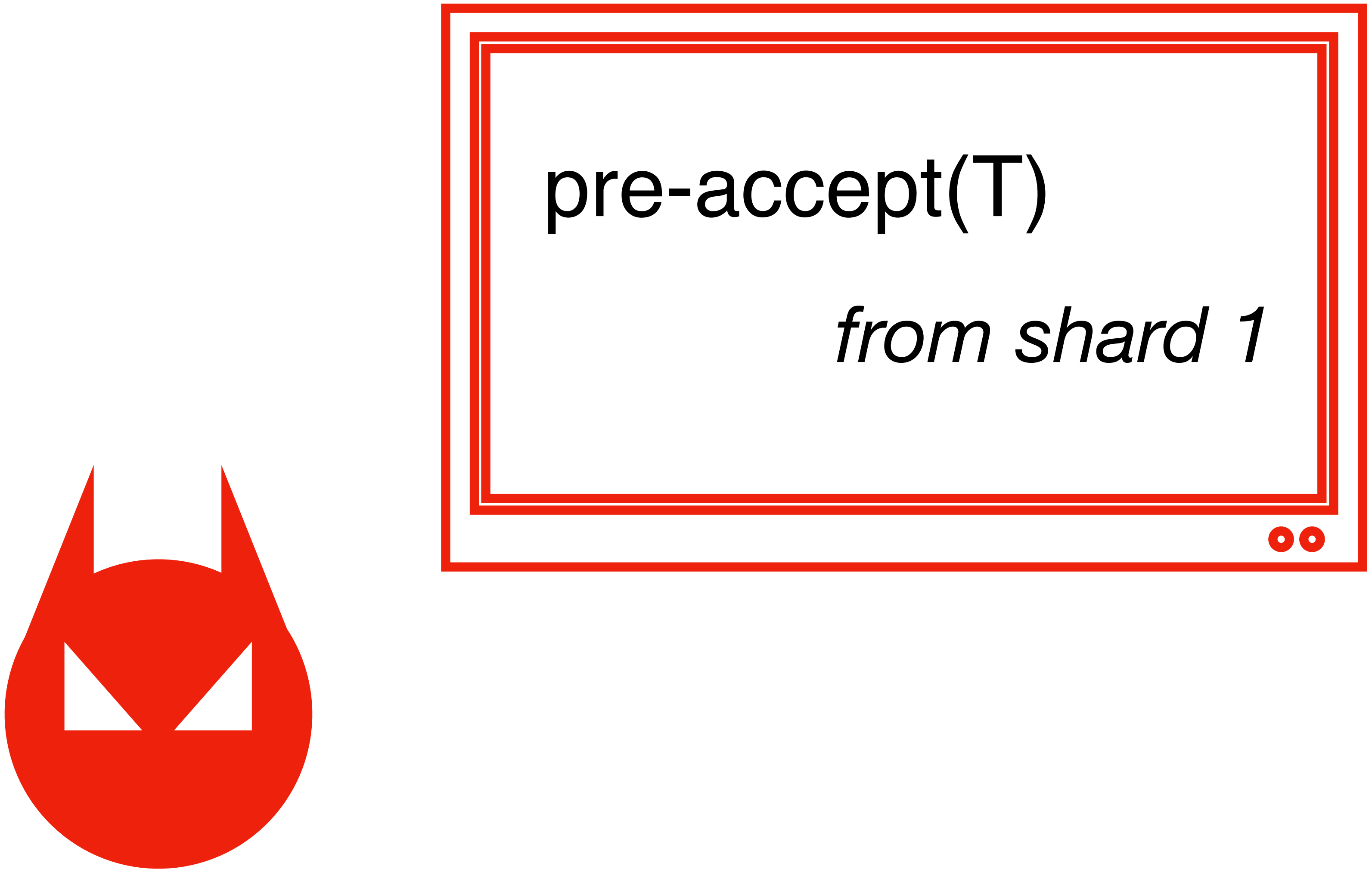
transaction

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



Client-Led Cross-Shard Consensus

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



pre-accept(T)

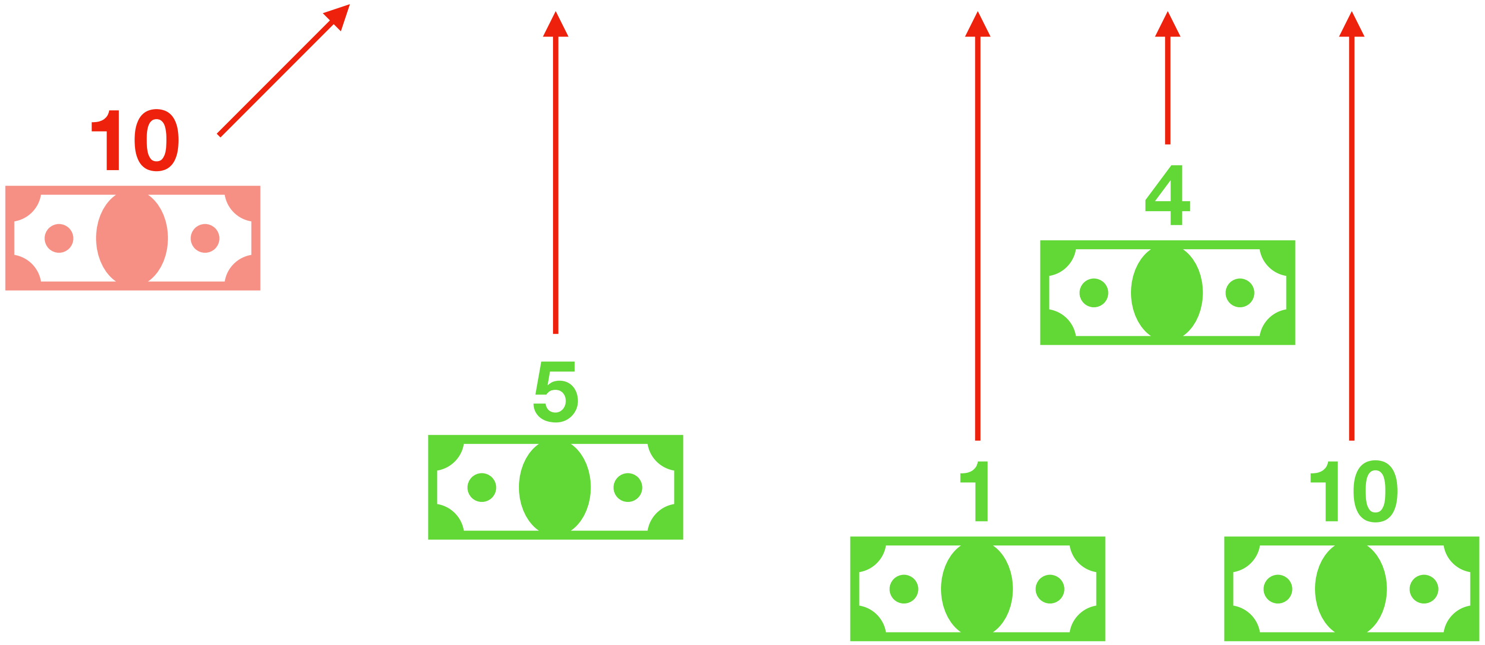
from shard 1

Client-Led Cross-Shard Consensus

- First phase attacks: let's double-spend X₁

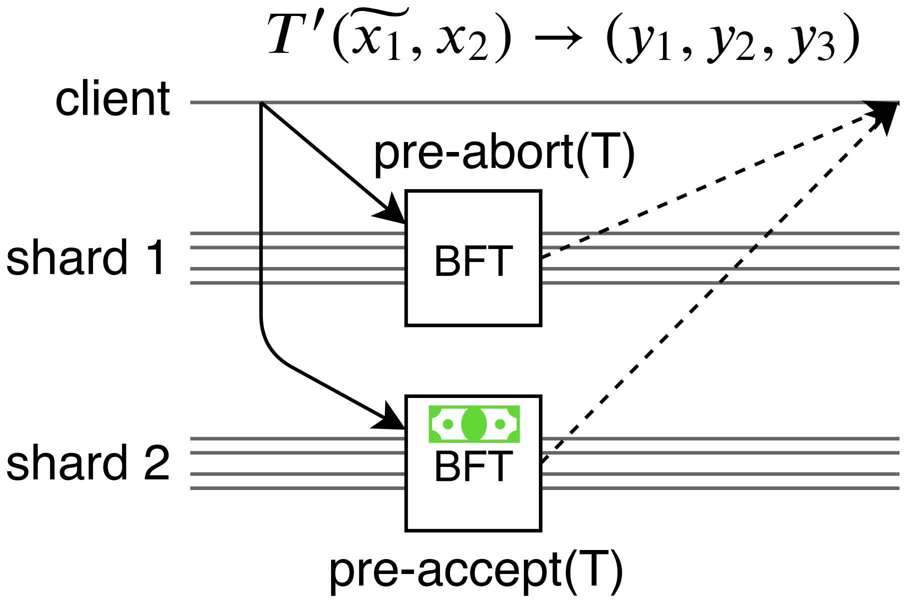
(bad) transaction

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



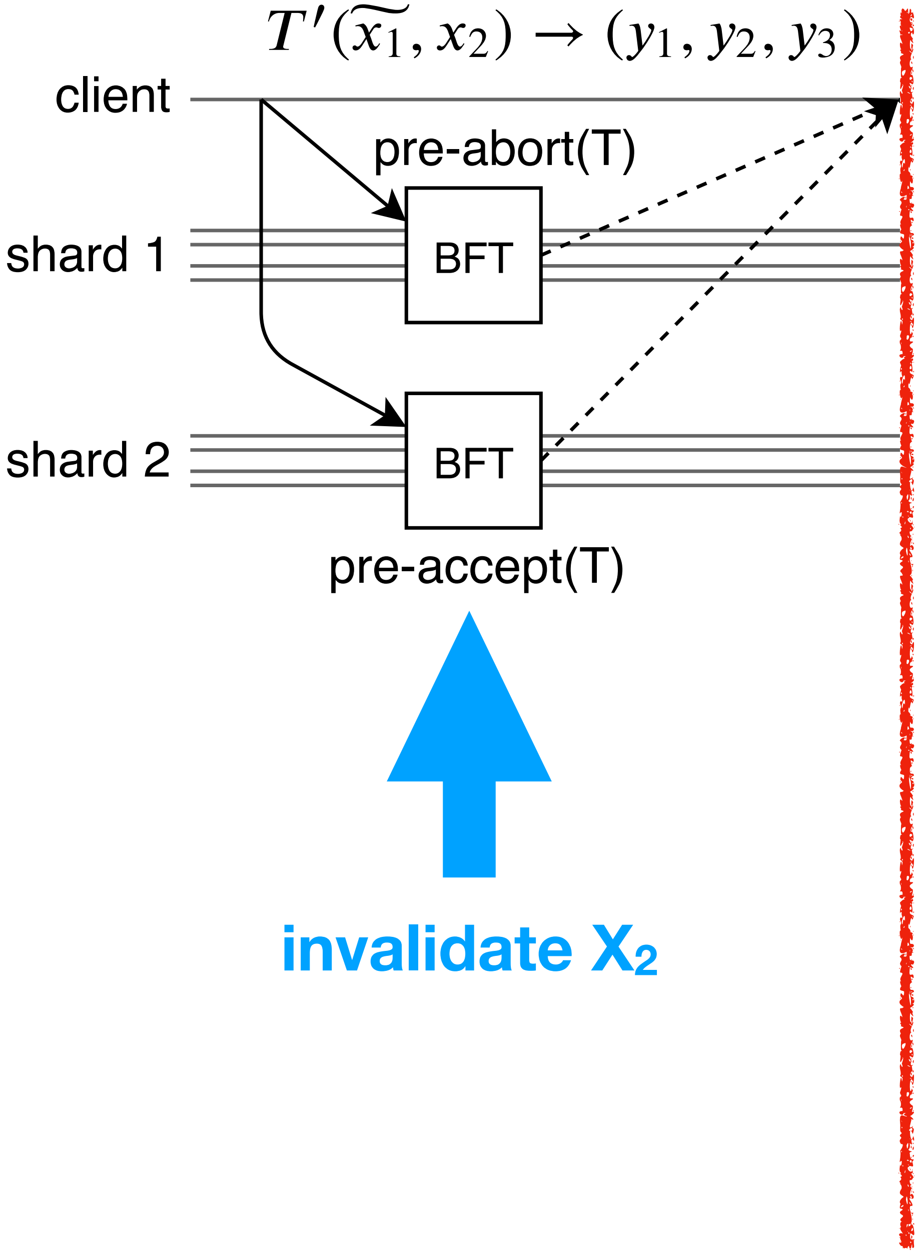
Client-Led Cross-Shard Consensus

- First phase attacks: recording messages



Client-Led Cross-Shard Consensus

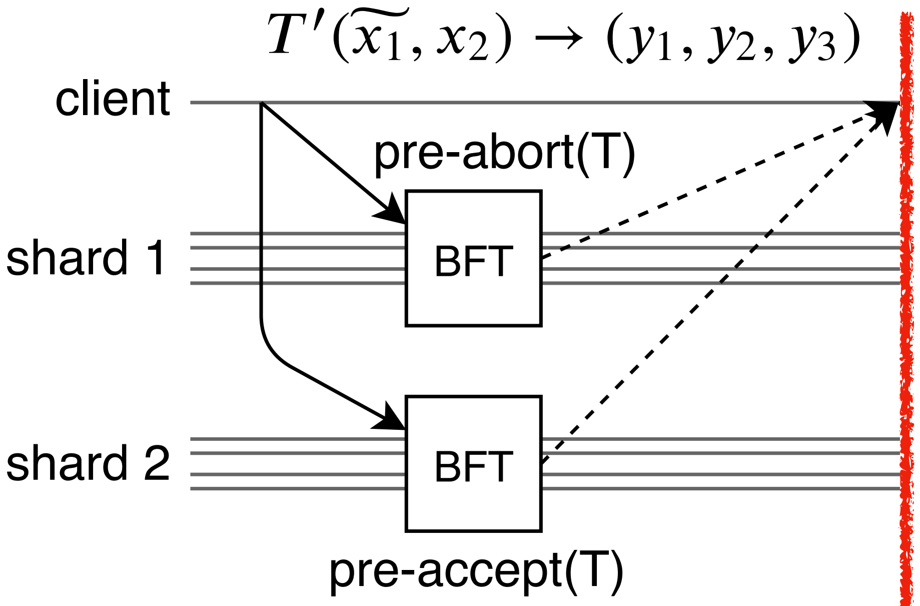
- First phase attacks: recording messages



invalidate X_2

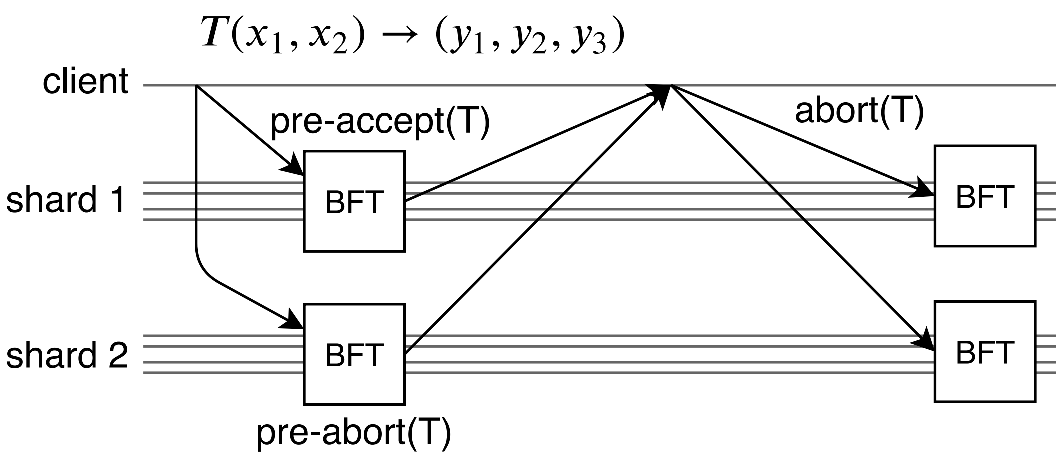
Client-Led Cross-Shard Consensus

- First phase attacks: recording messages



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

invalidate X_2

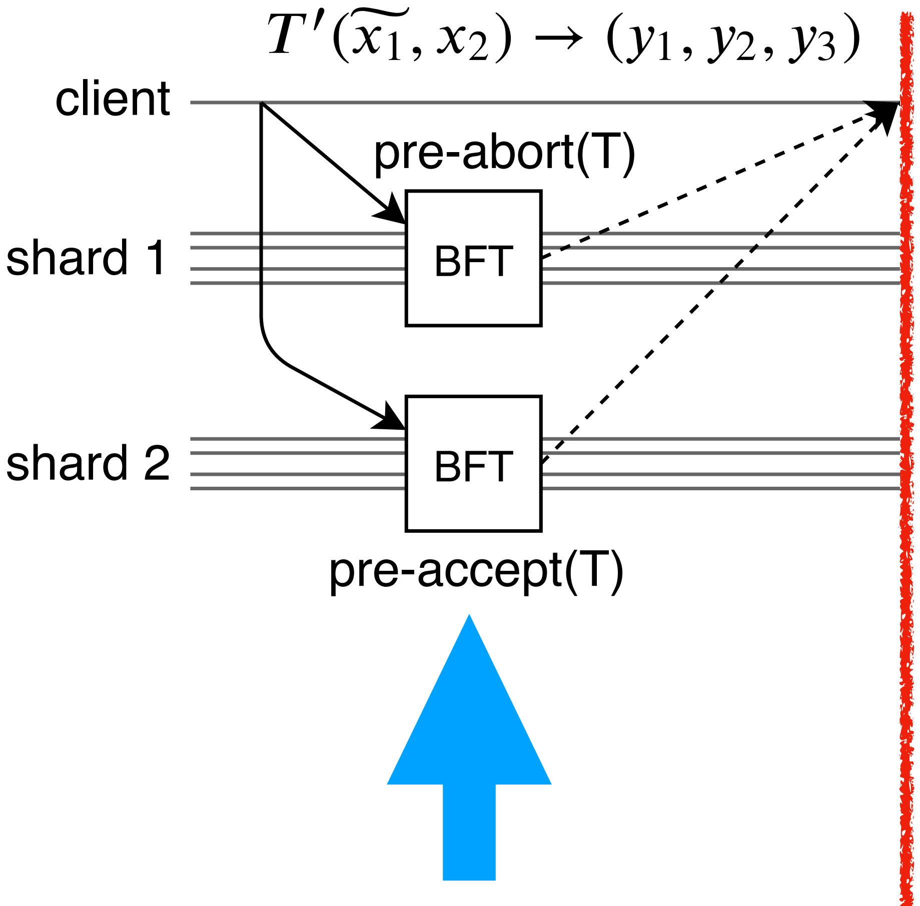


$T(x_1, x_2) \rightarrow (y_1, y_2, y_3)$

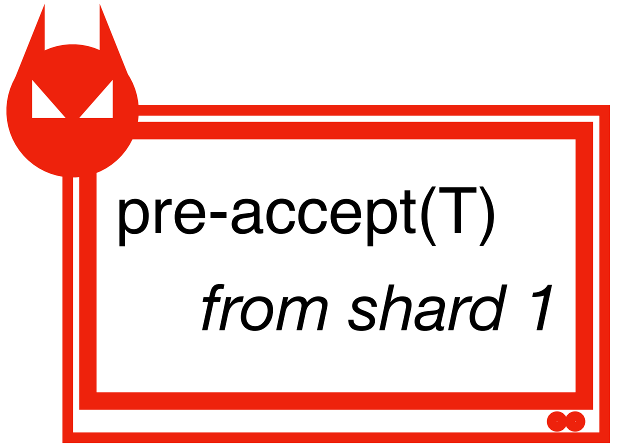
(because X_2 is invalidated)

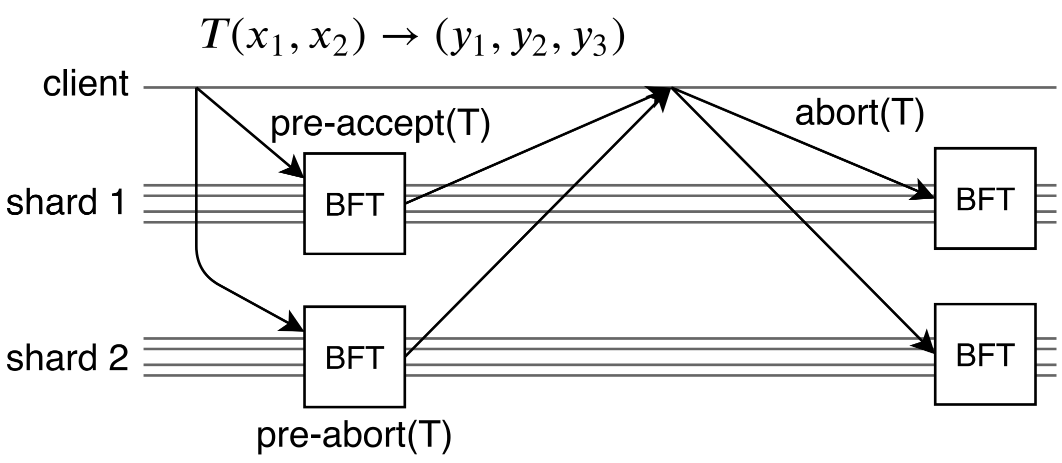
Client-Led Cross-Shard Consensus

- First phase attacks: recording messages



invalidate X_2

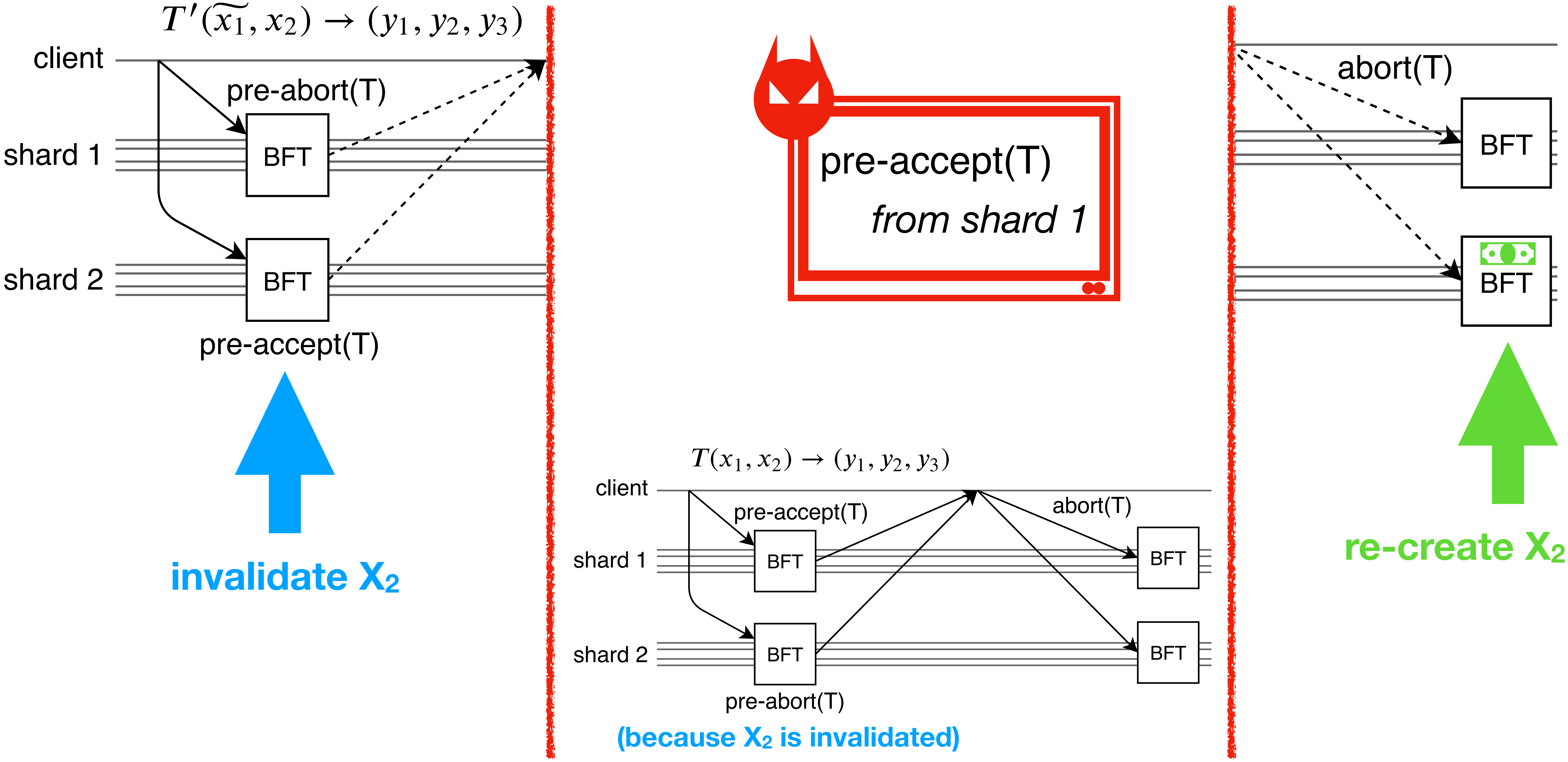




(because X_2 is invalidated)

Client-Led Cross-Shard Consensus

- First phase attacks: recording messages



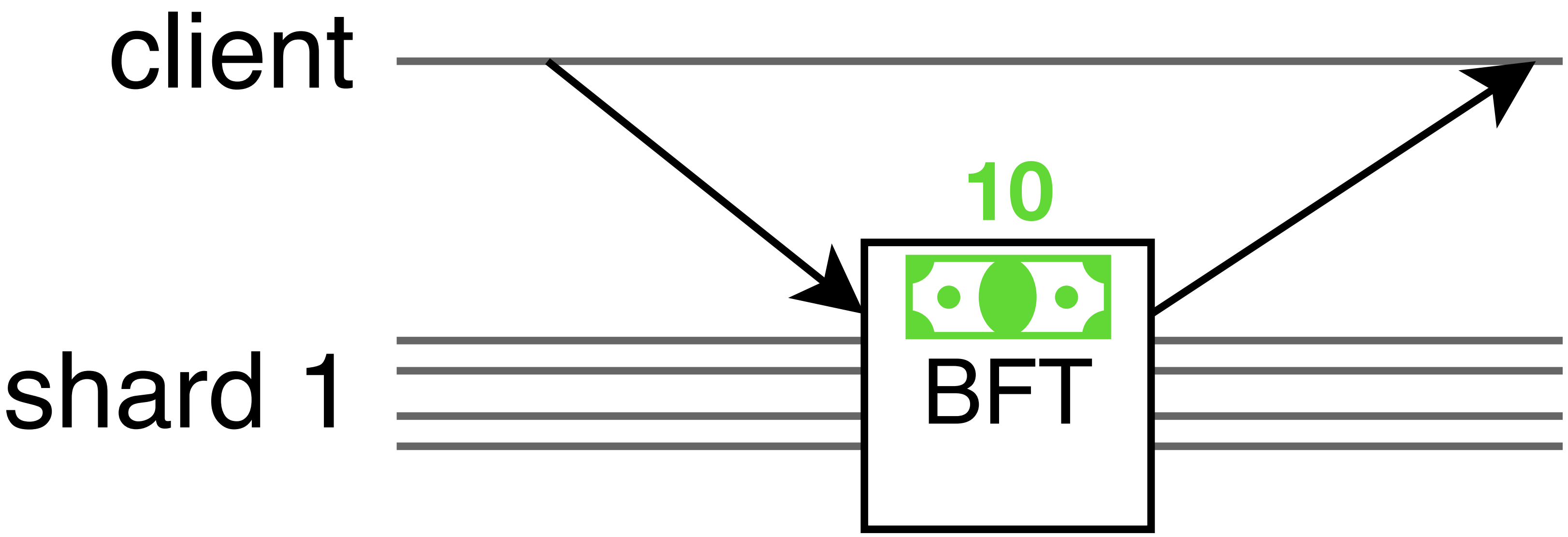
Client-Led Cross-Shard Consensus

- First phase attacks: spend X1

pre-accept(T)

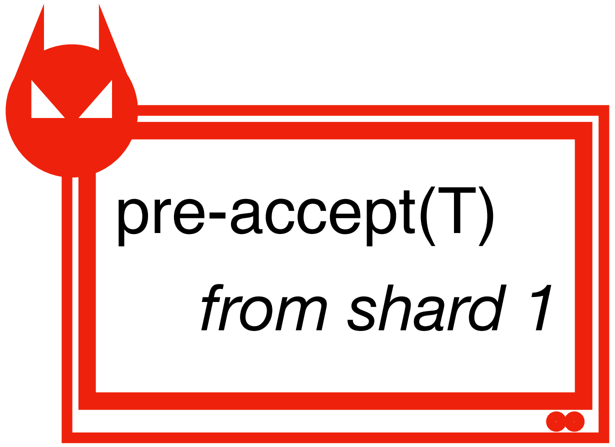
from shard 1

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



Client-Led Cross-Shard Consensus

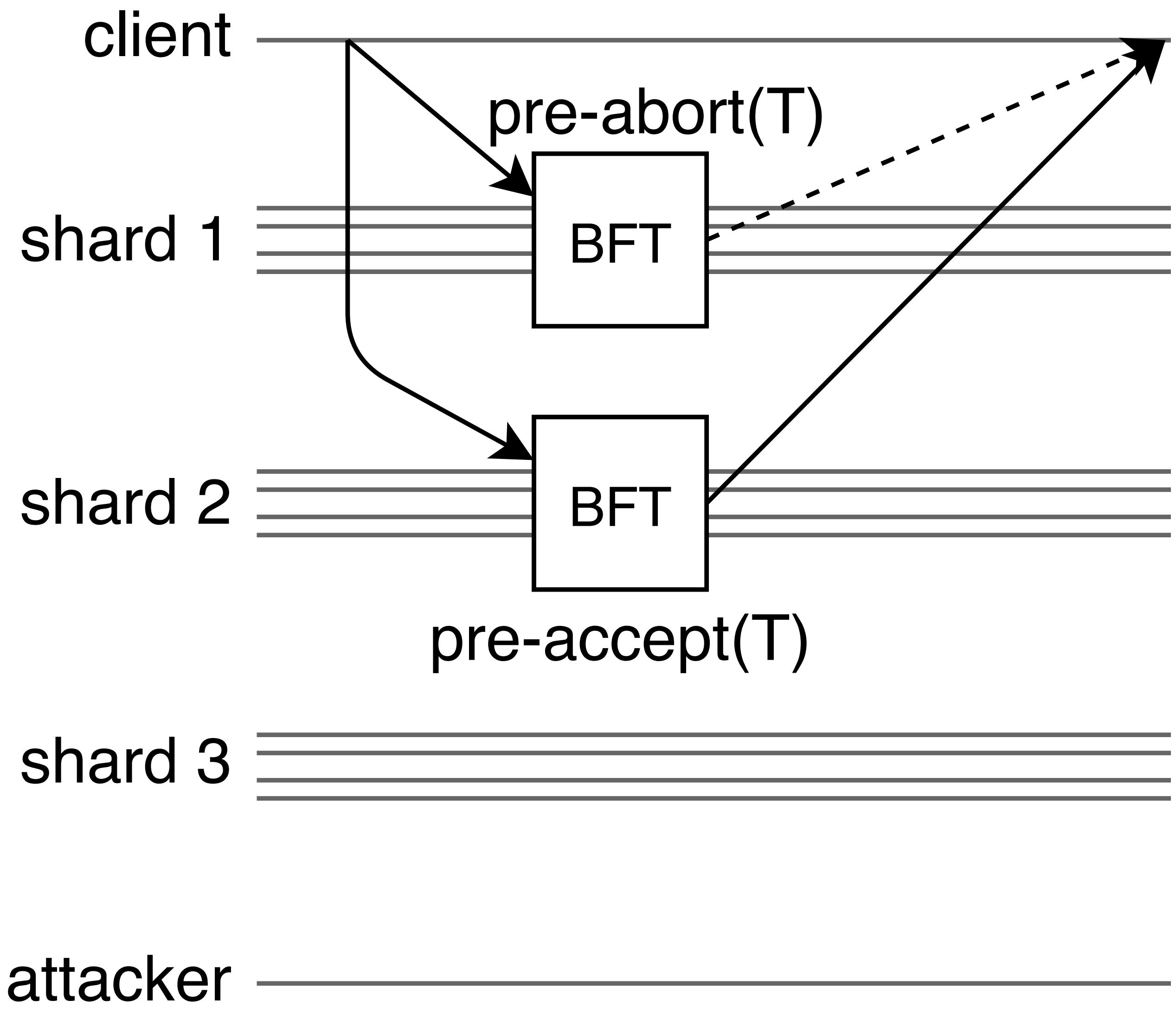
- First phase attacks: double-spend X_1



pre-accept(T)

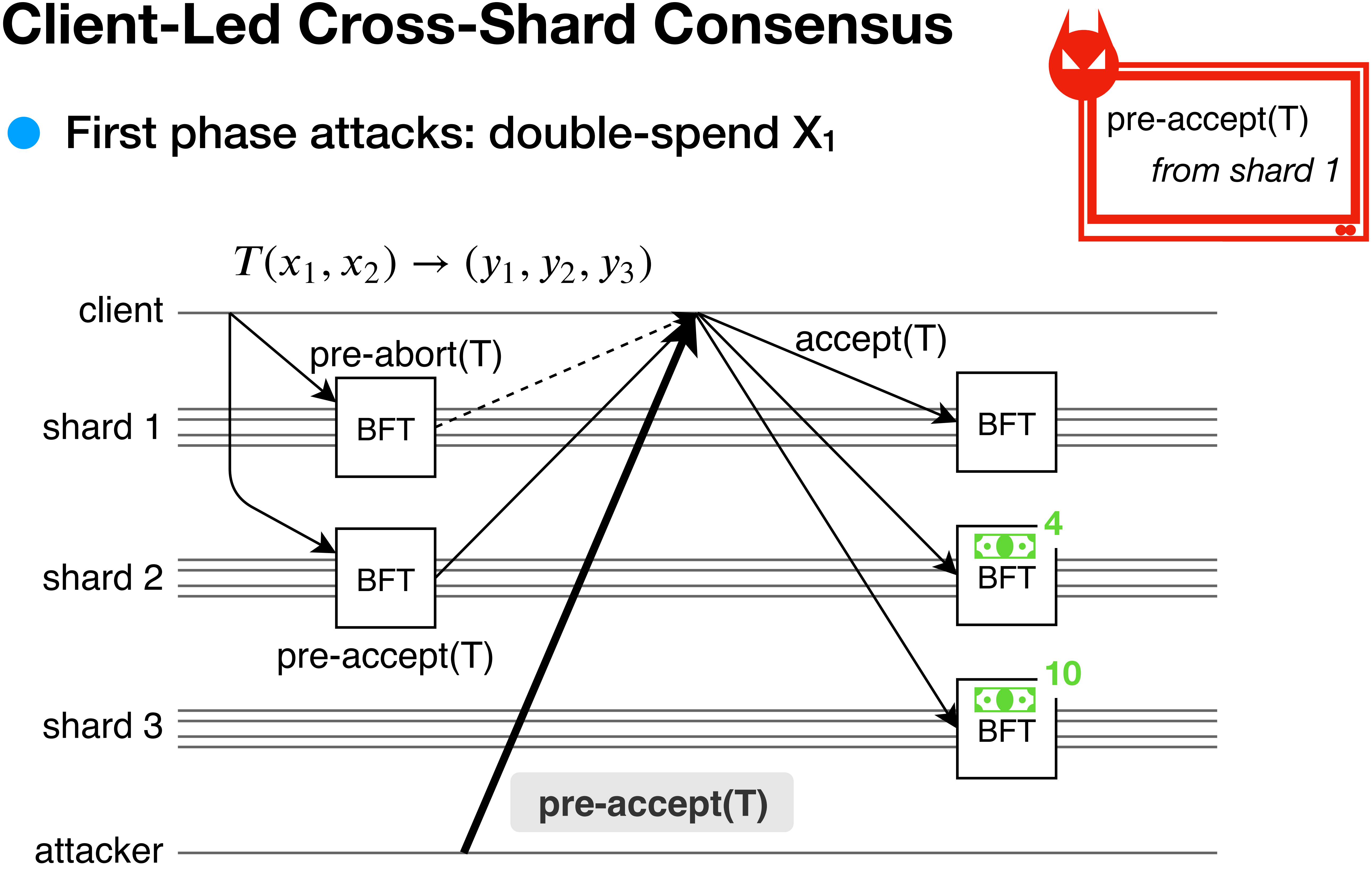
from shard 1

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



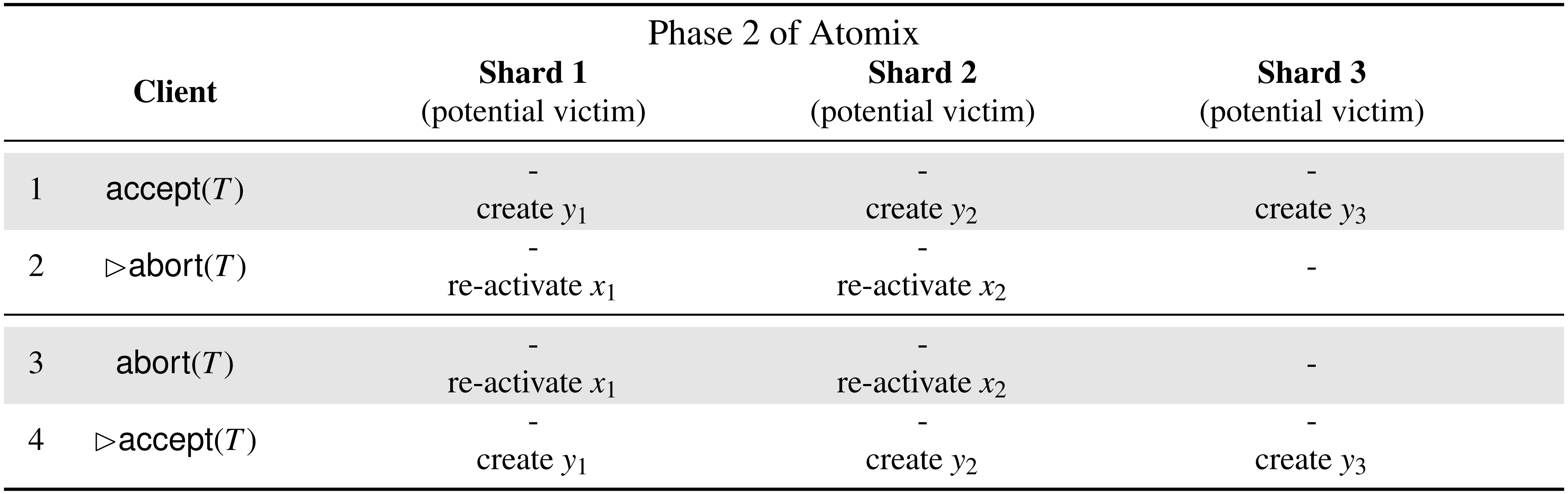
Client-Led Cross-Shard Consensus

- First phase attacks: double-spend X_1



Client-Led Cross-Shard Consensus

- Second phase attacks

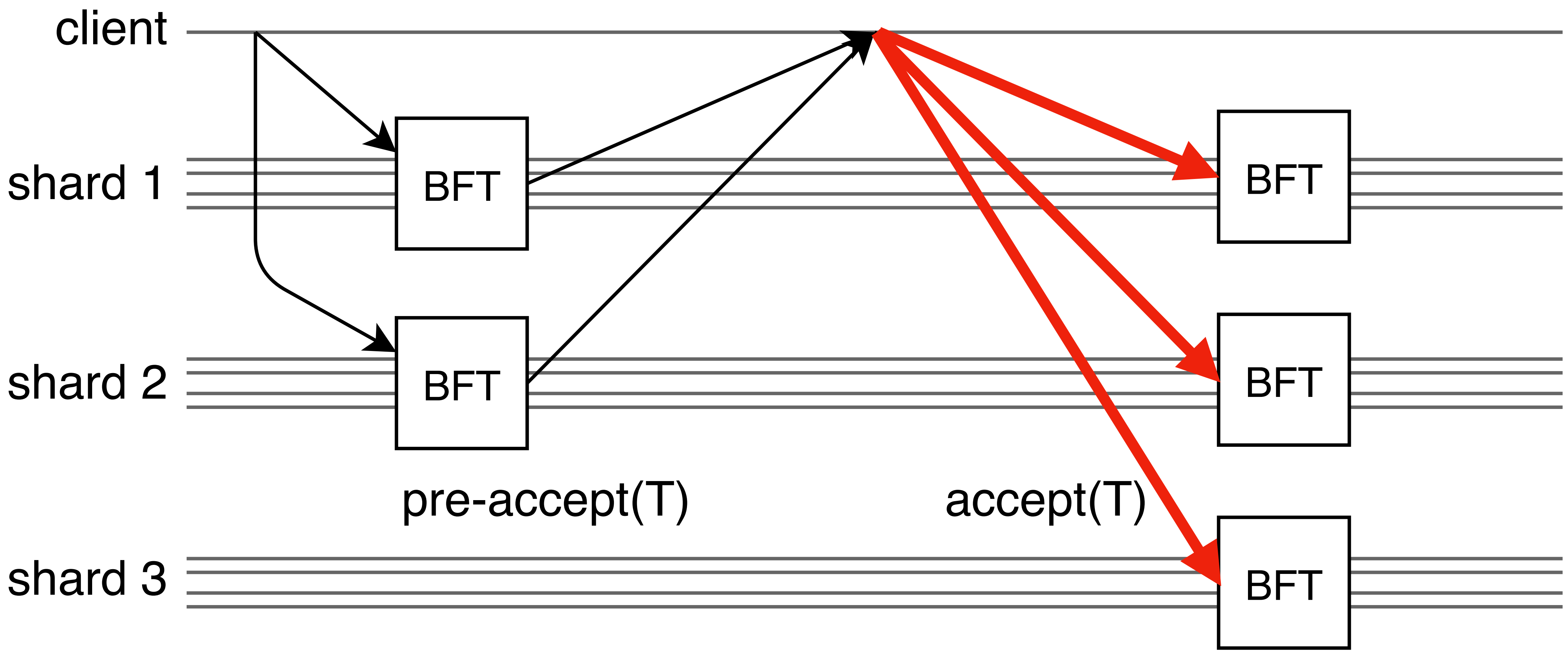


		Phase 2 of Atomix		
	Client	**Shard 1** (potential victim)	**Shard 2** (potential victim)	**Shard 3** (potential victim)
1	accept(T)	- create y_1	- create y_2	- create y_3
2	▷abort(T)	- re-activate x_1	- re-activate x_2	-
3	abort(T)	- re-activate x_1	- re-activate x_2	-
4	▷accept(T)	- create y_1	- create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

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

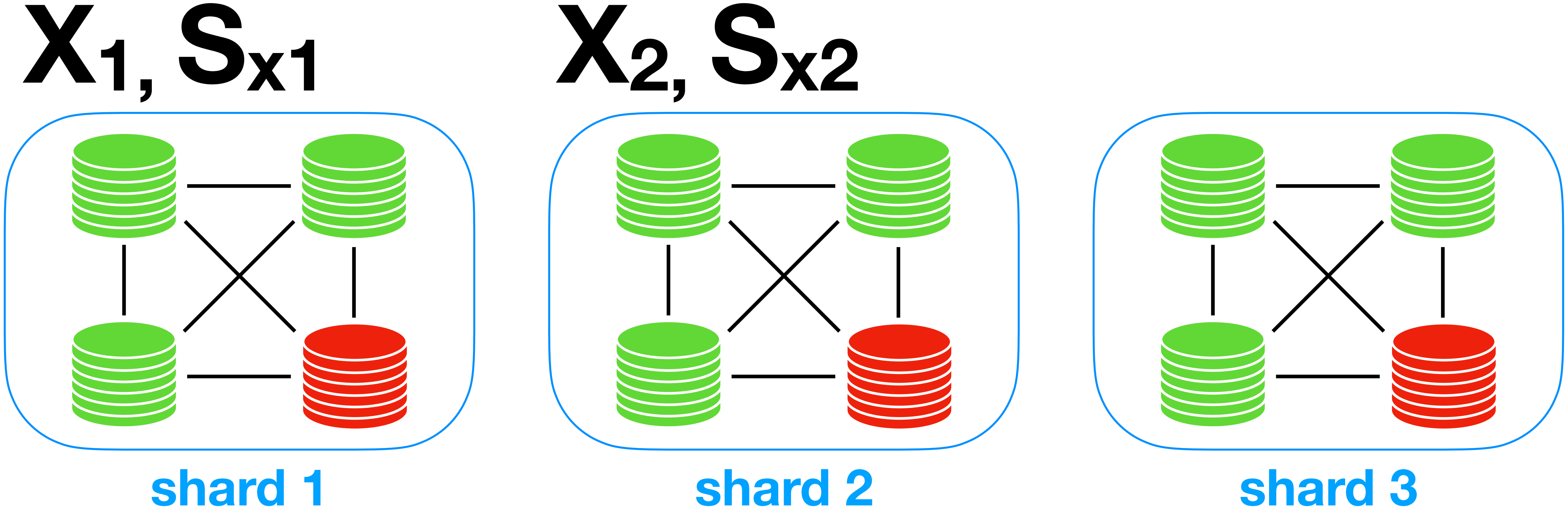
Fixing replay attacks without breaking scalability

Byzcuit

Chainspace ❤ Omniledger

Byzcuit

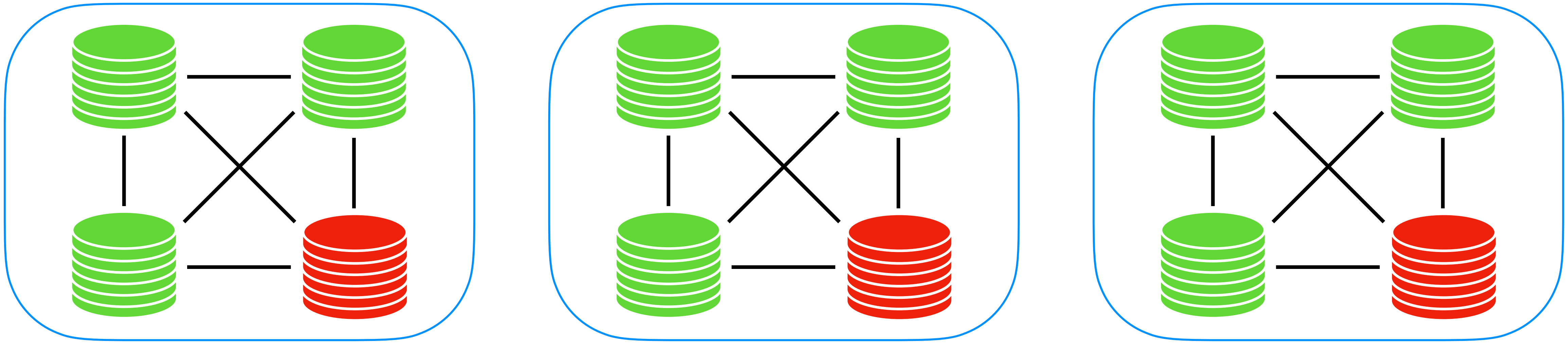
- Fixing issue 1: adding sequence numbers per object



Byzcuit

- Fixing issue 2: dummy objects for output shards

X_1, S_{x1} X_2, S_{x2} D_3, S_{D3}



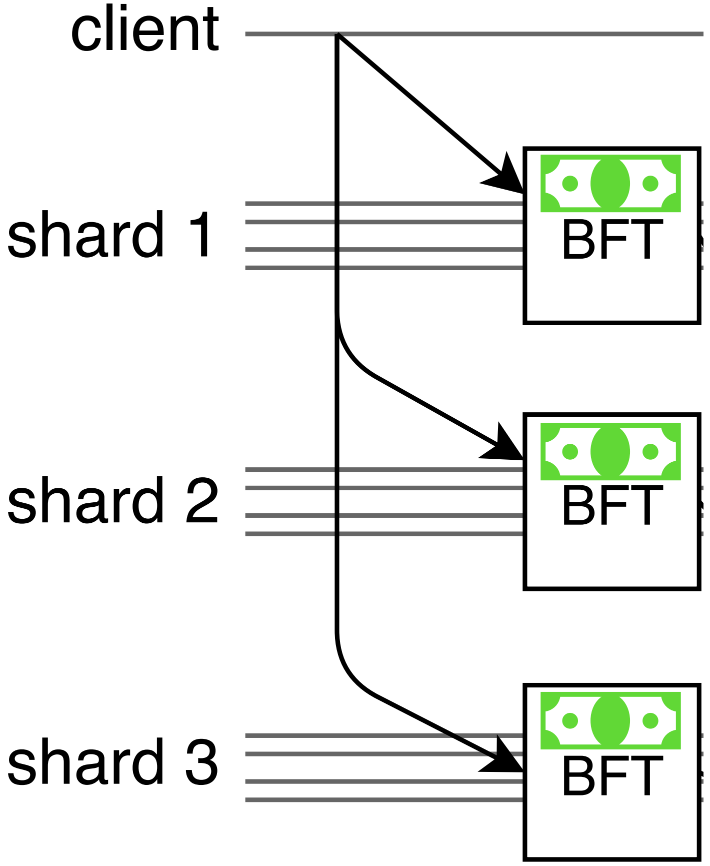
shard 1 shard 2 shard 3

Byzcuit

$$\left\{s_T, T(x_1, x_2) \rightarrow (y_1, y_2, y_3)\right\}$$

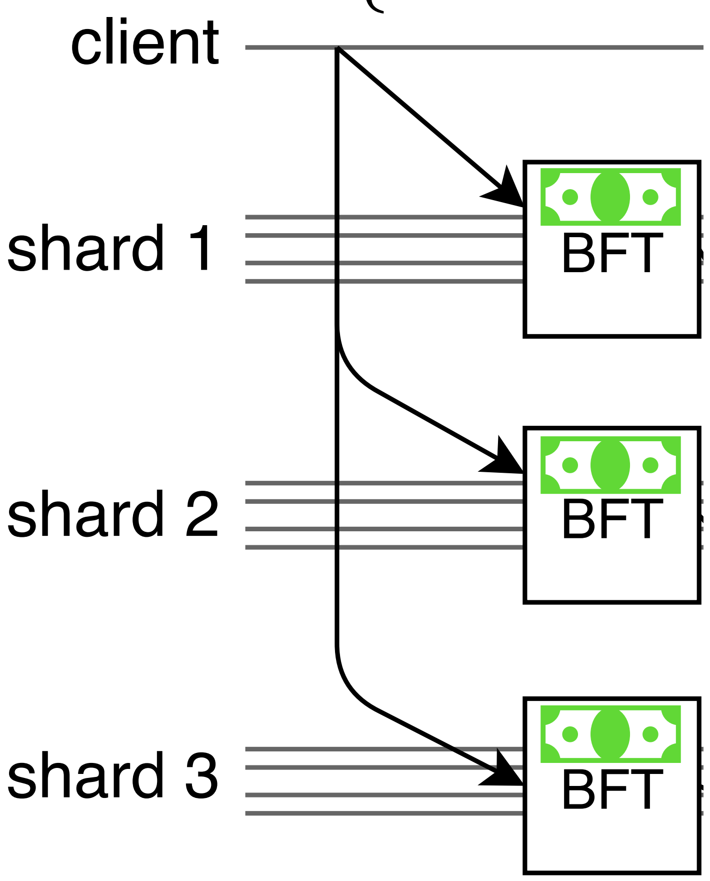
Byzcuit

$$\left\{s_T, T(x_1, x_2) \rightarrow (y_1, y_2, y_3)\right\}$$



Byzcuit

$$\left\{s_T, T(x_1, x_2) \rightarrow (y_1, y_2, y_3)\right\}$$

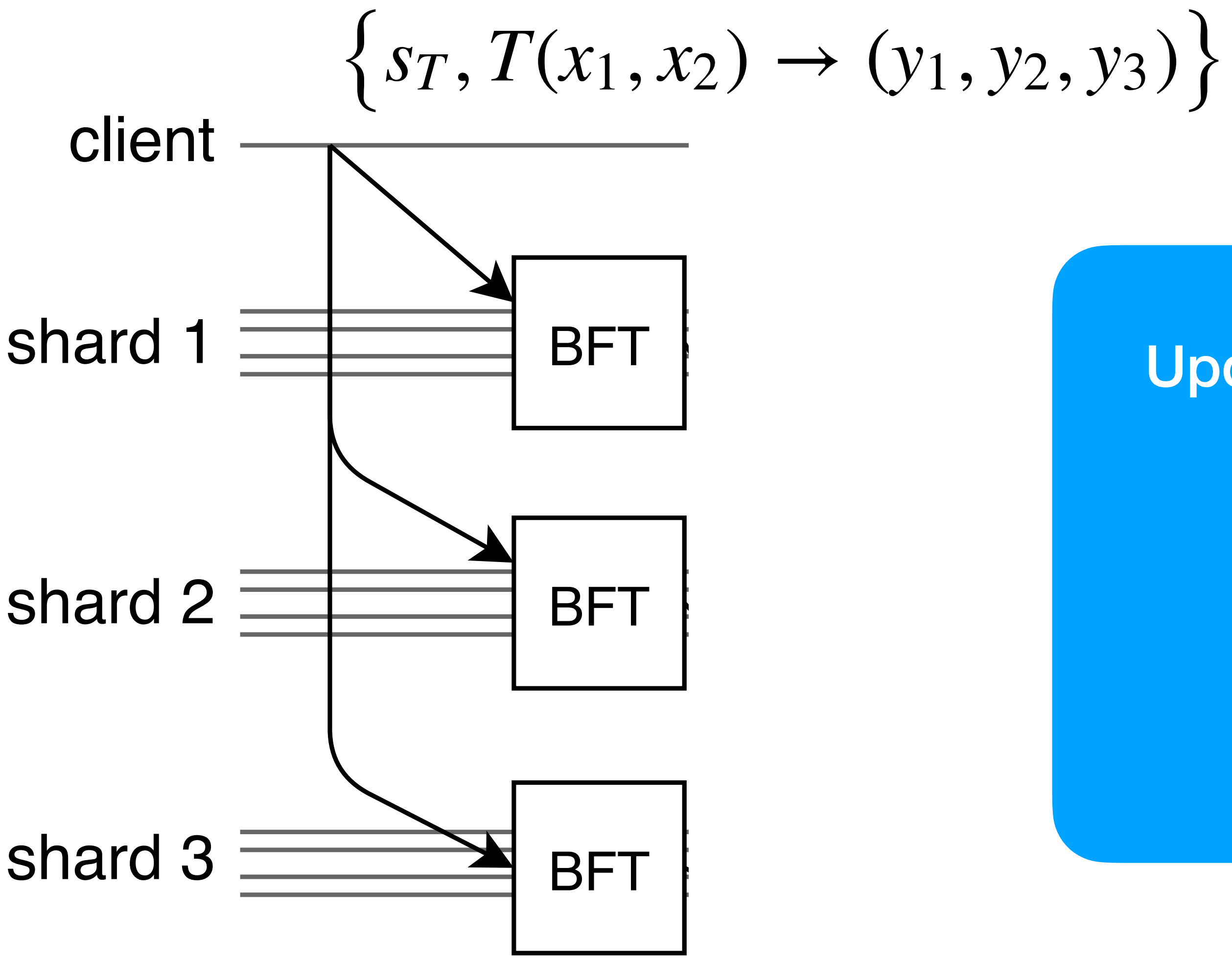


Check 1. Are all inputs active / transaction well formed ?

Check 2. Is the sequence number S$_T$

$S_T \geq max\{S_{X1}, S_{X2}\}$ **?**

Byzcuit


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

If Check fail:
Update all sequence numbers

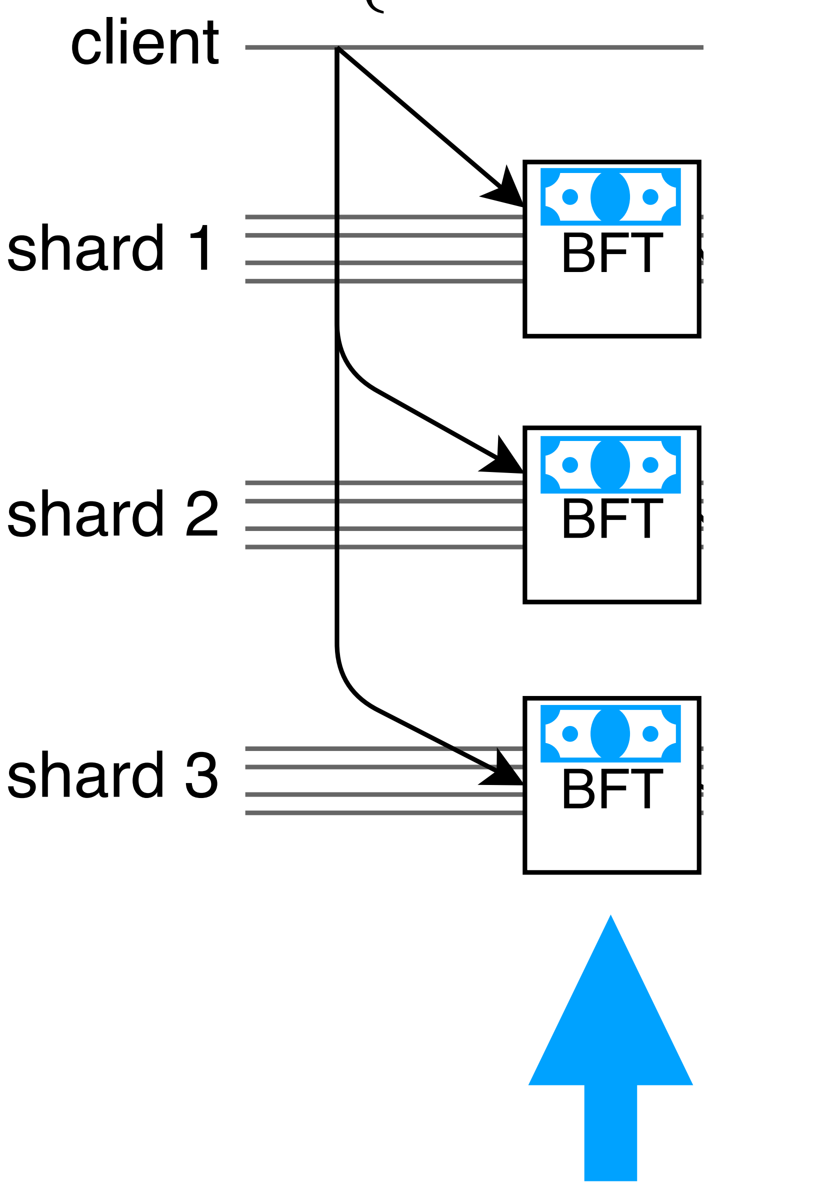
$$S_{X1} \leftarrow S_T + 1$$

$$S_{X2} \leftarrow S_T + 1$$

$$S_{D3} \leftarrow S_T + 1$$

Byzcuit

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

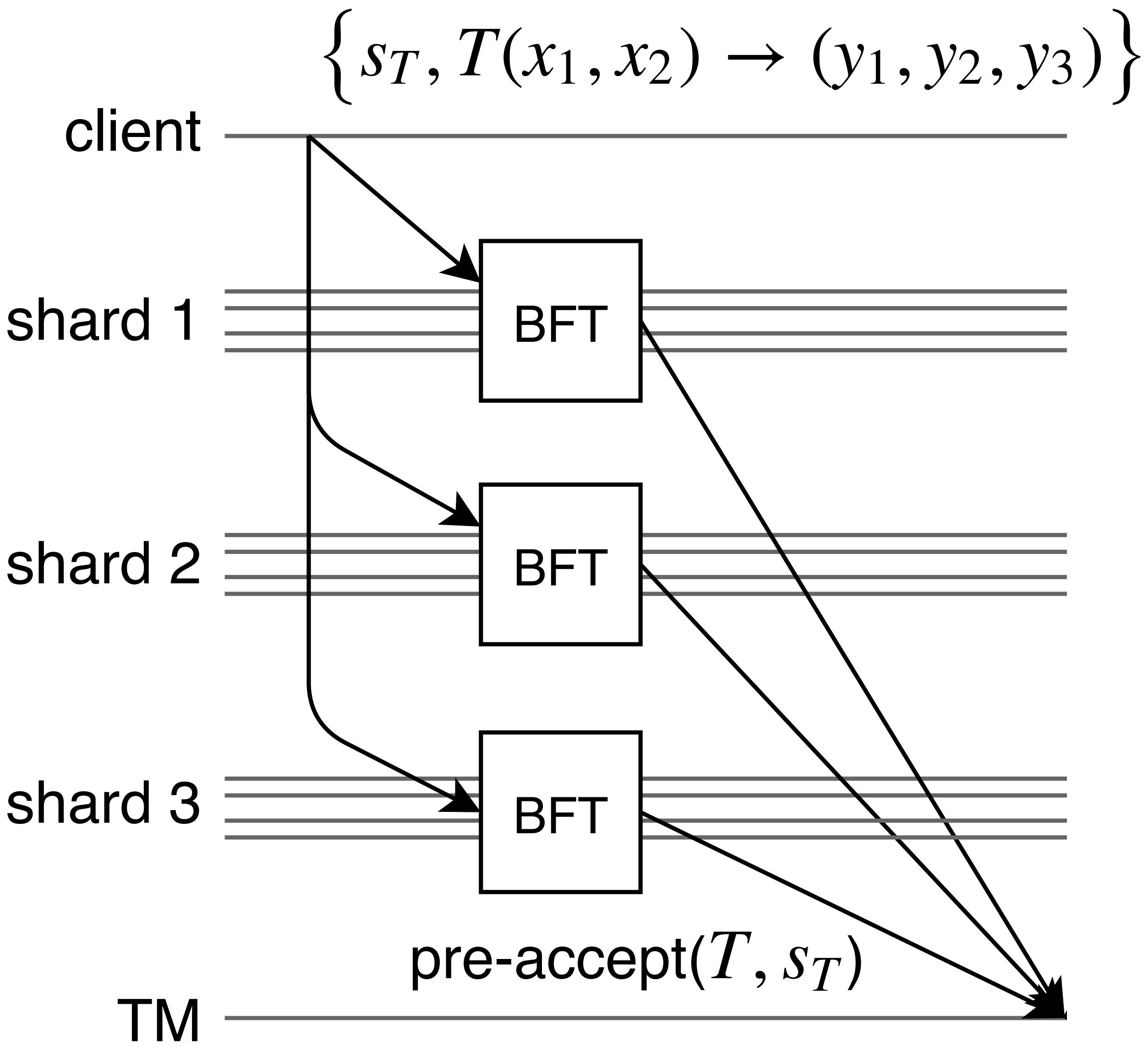


lock X_1, X_2, D_3

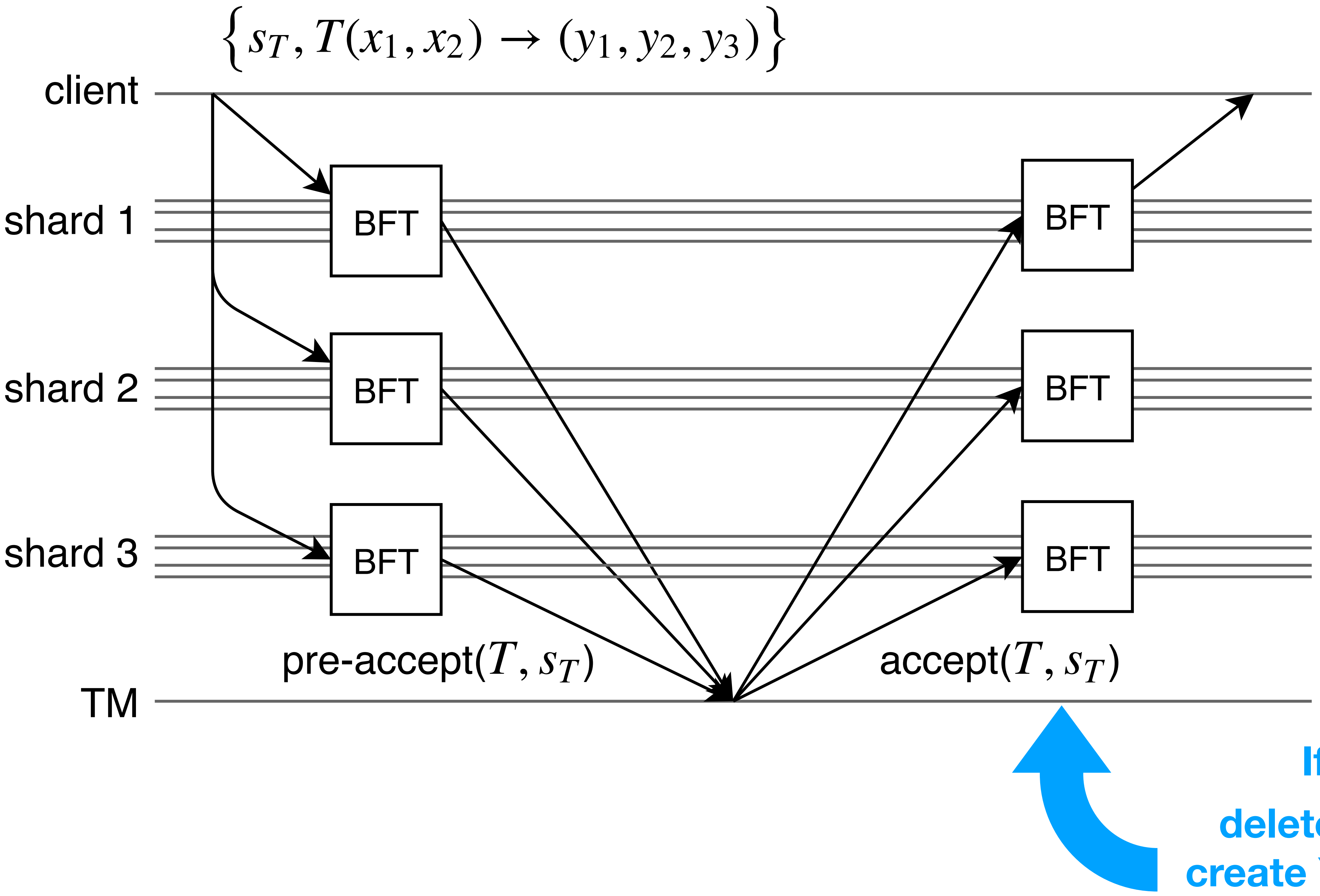
If Check pass:

1. Lock objects as Chainspace
2. Store the session ID (S_T, T)

Byzcuit


$\left\{s_T, T(x_1, x_2) \rightarrow (y_1, y_2, y_3)\right\}$
client
shard 1
BFT
shard 2
BFT
shard 3
BFT
pre-accept(T, s_T)
TM

Byzcuit

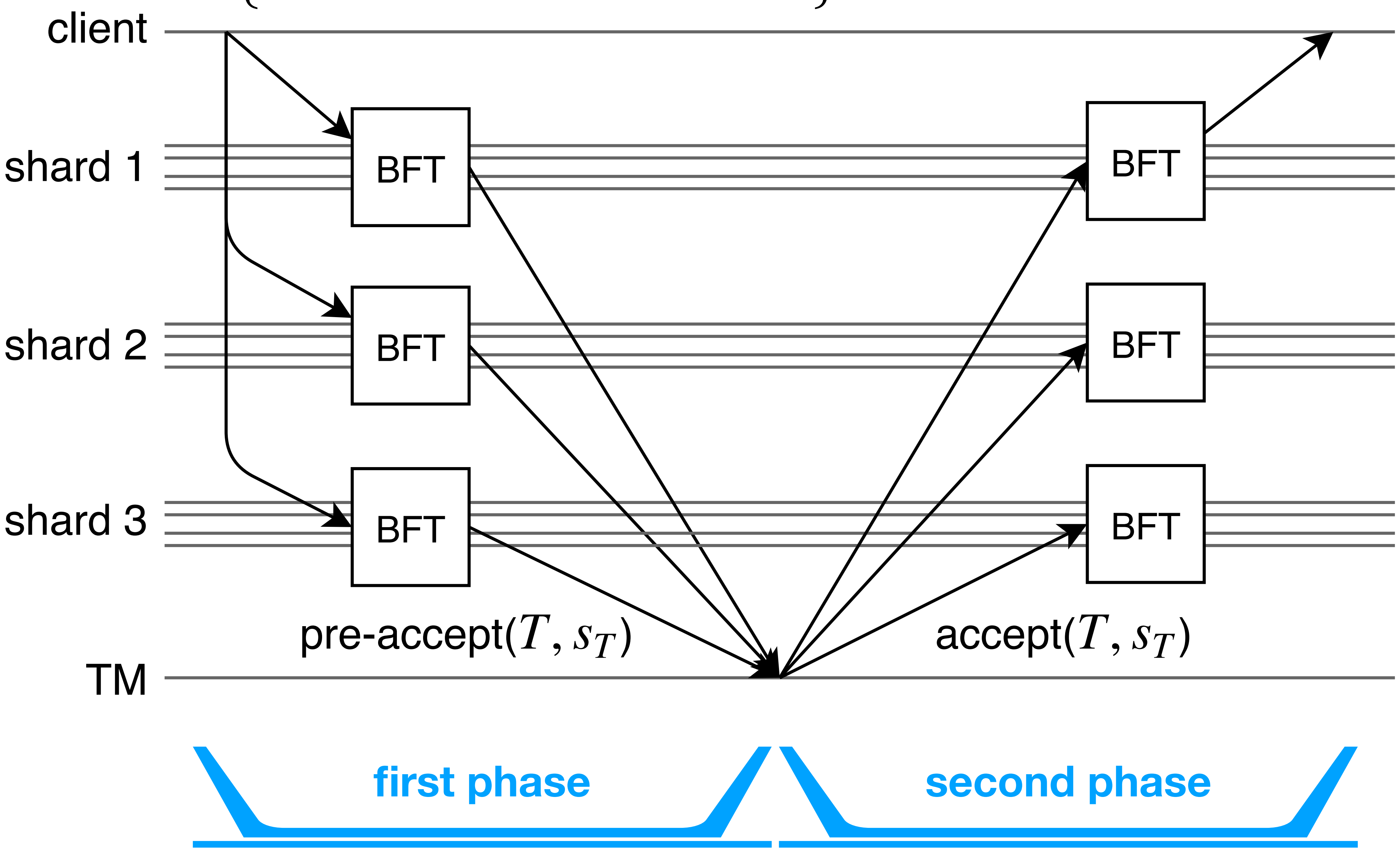


If (S_T, T)

delete X_1, X_2, D_3
create Y_1, Y_2, Y_3, $D_{3'}$

Byzcuit

$$\left\{ s_T, T(x_1, x_2) \rightarrow (y_1, y_2, y_3) \right\}$$



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.

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

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

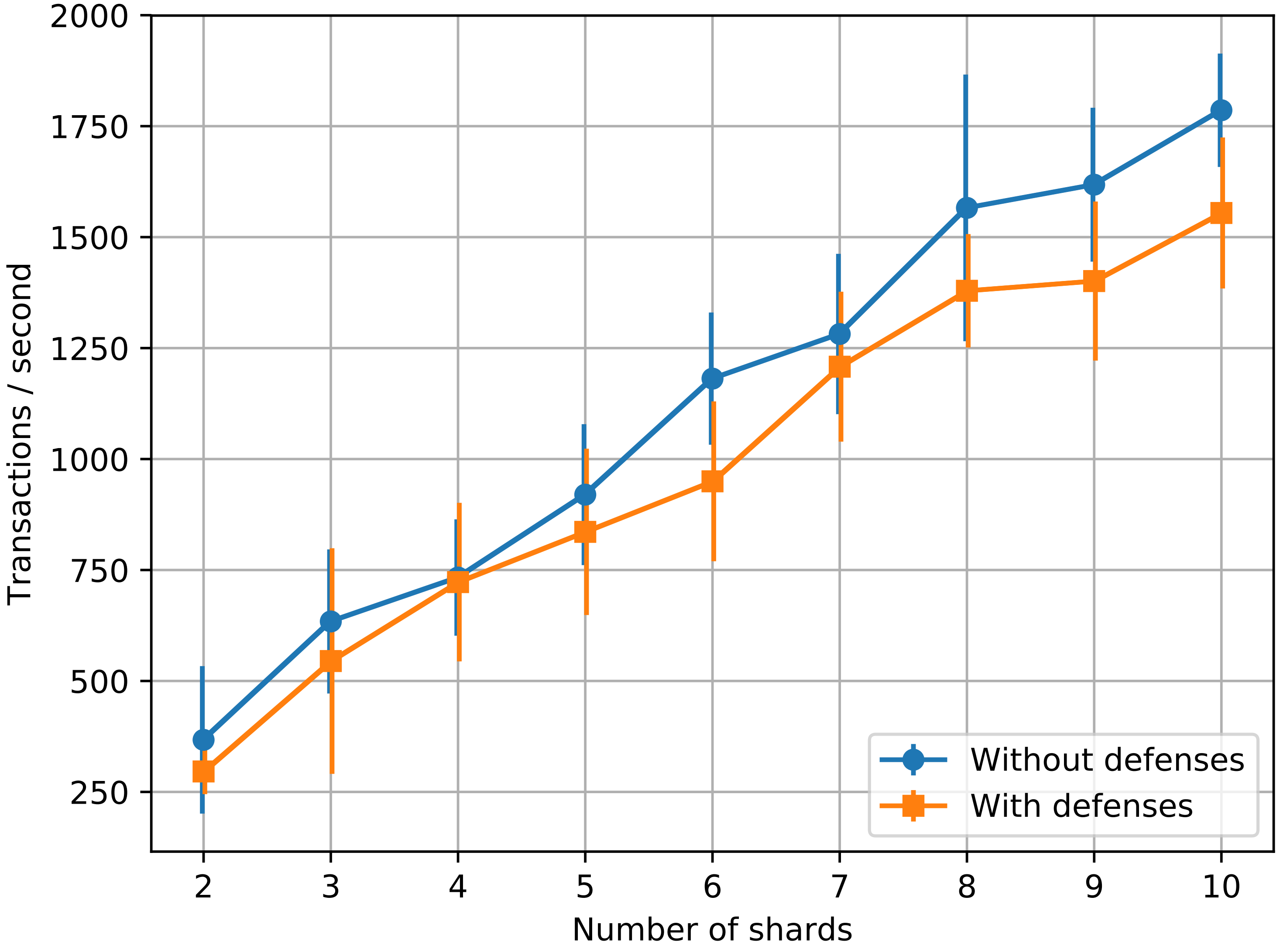
- Performance

Open Source

https://github.com/sheharbano/byzcuit

Byzcuit

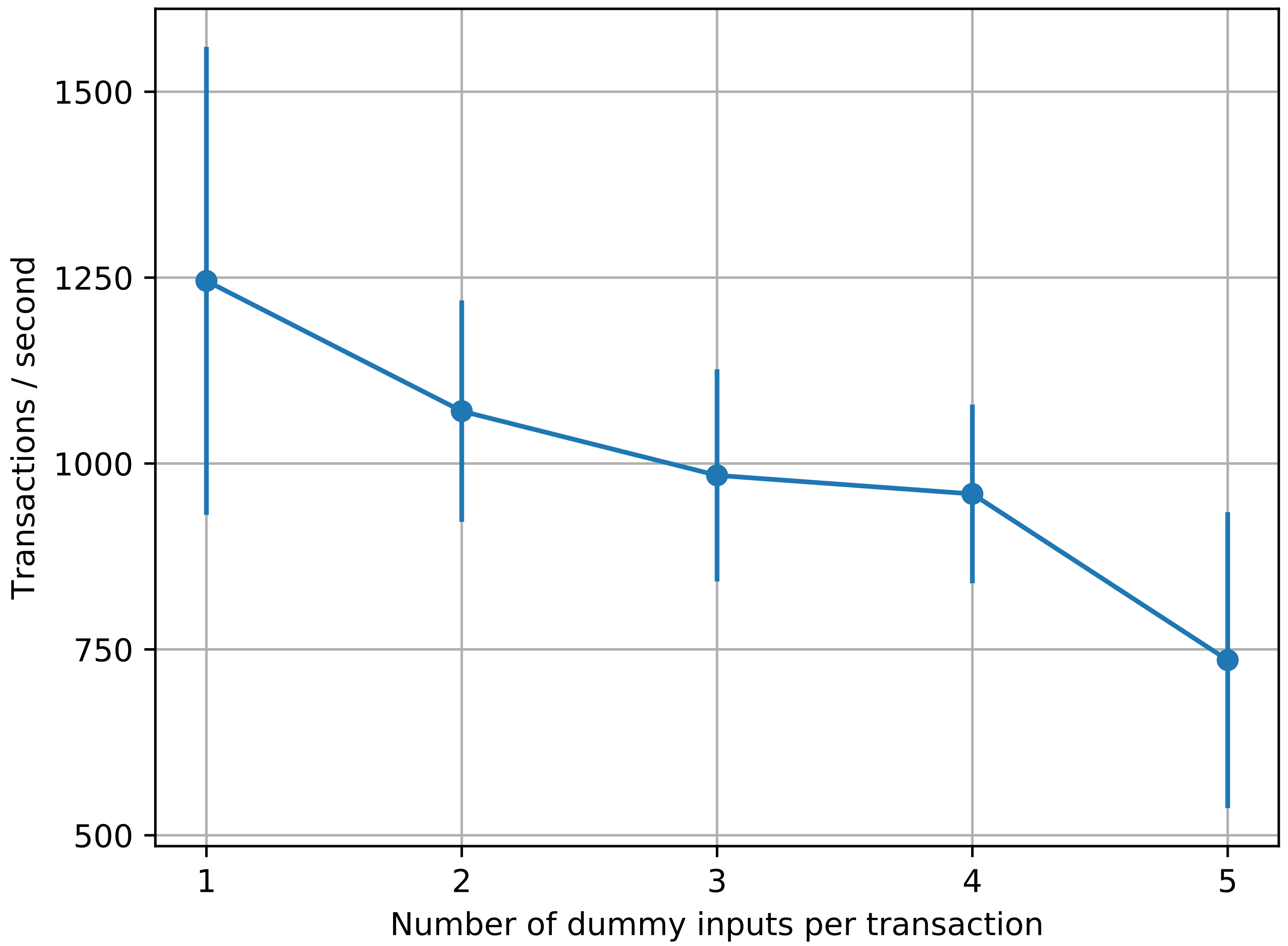
- Performance



(2 inputs ; 5 outputs)

Byzcuit

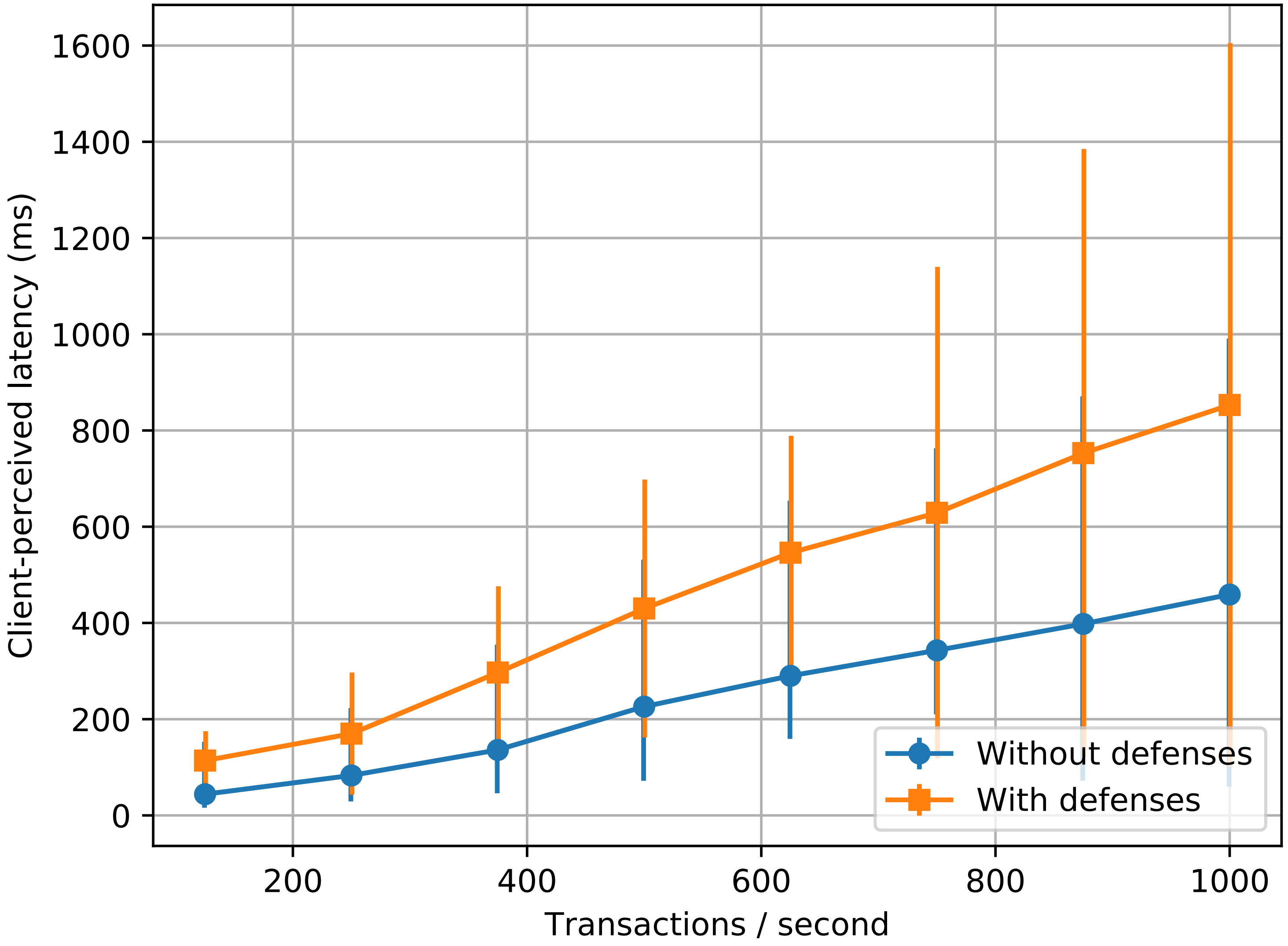
- Performance



(1 input ; 6 shards)

Byzcuit

- Performance

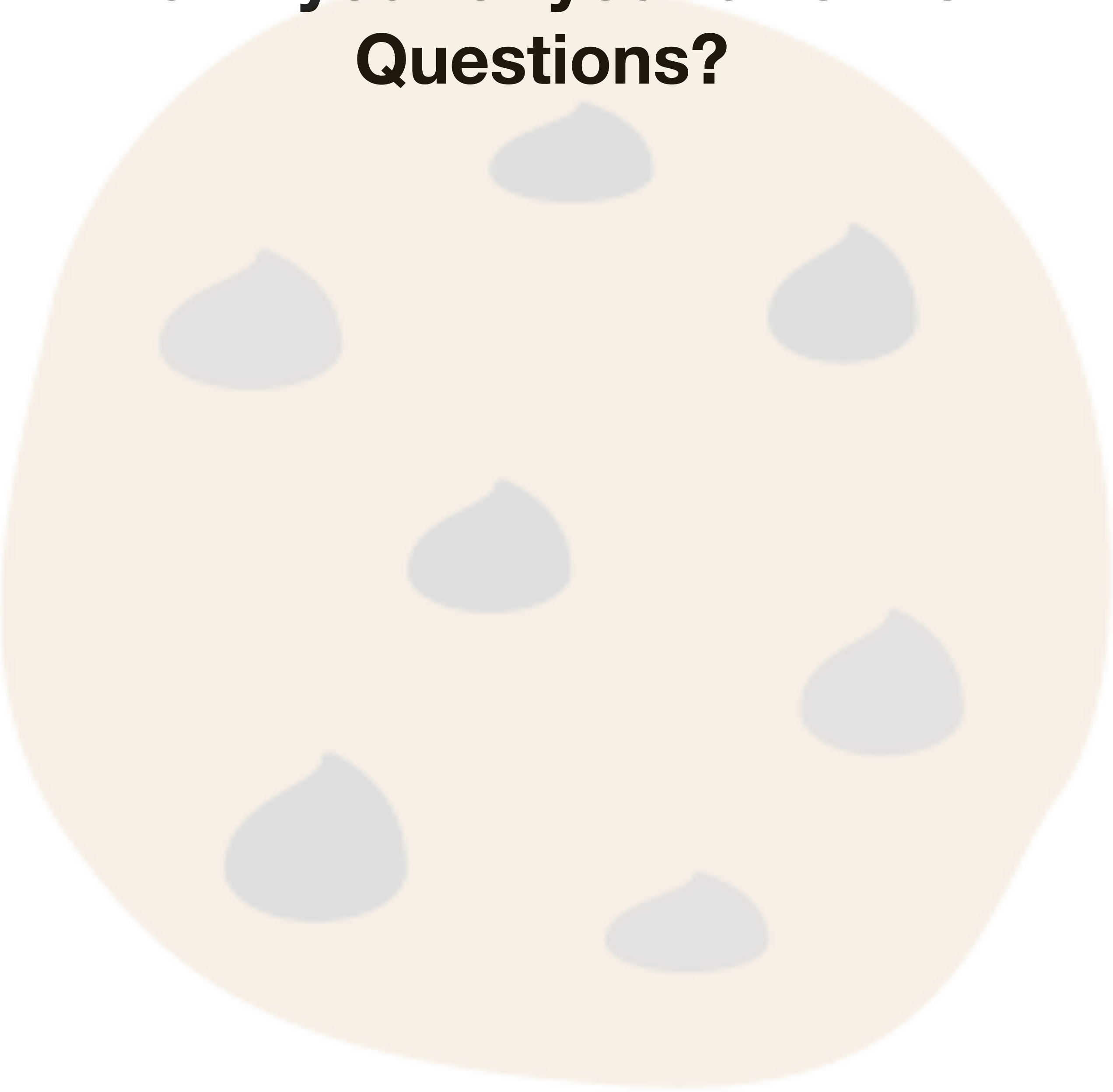


(2 input ; 5 outputs ; 6 shards)

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?



Alberto Sonnino
http://sonnino.com

This work is supported in part by EPSRC Grant EP/M013286/1, the EU H2020 DECODE project (grant agreement number 732546), and chainspace.io