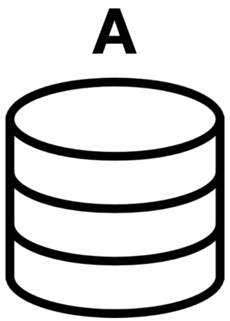
Practical Smart Contract Sharding with Static Program Analysis

Ilya Sergey

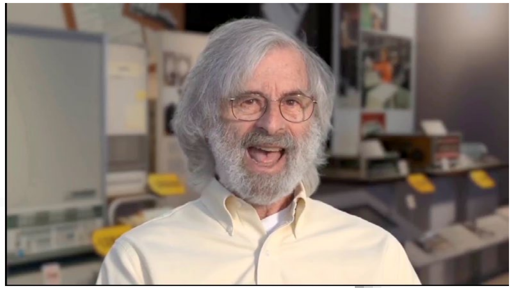
Workshop on Dependable and Secure Software Systems 2022

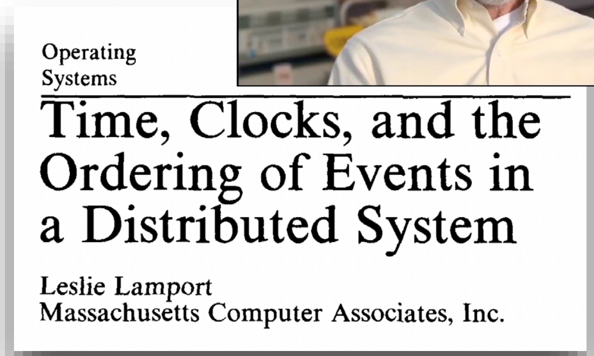
7 October 2022

A recipe for a robust distributed service



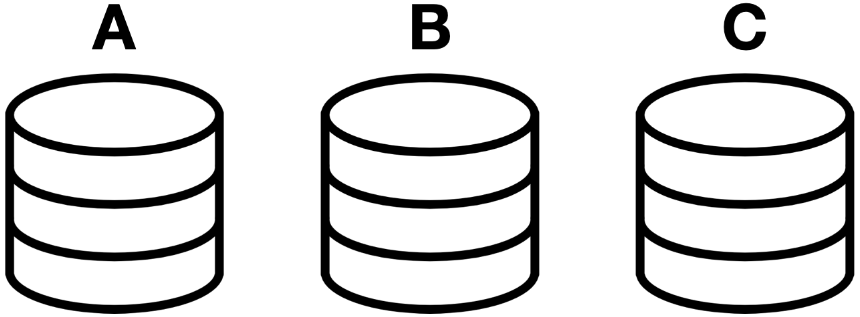
A recipe for a robust distributed service




Operating
Systems

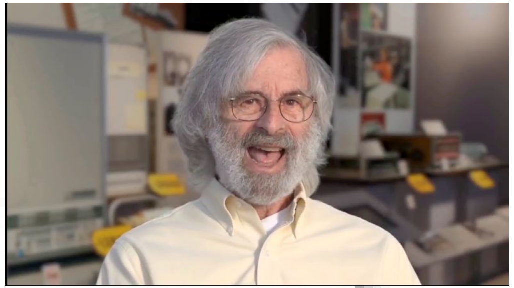
Time, Clocks, and the Ordering of Events in a Distributed System

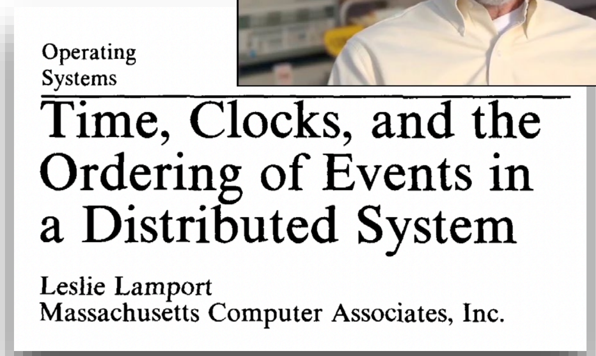
Leslie Lamport
Massachusetts Computer Associates, Inc.



1. Replicate your data

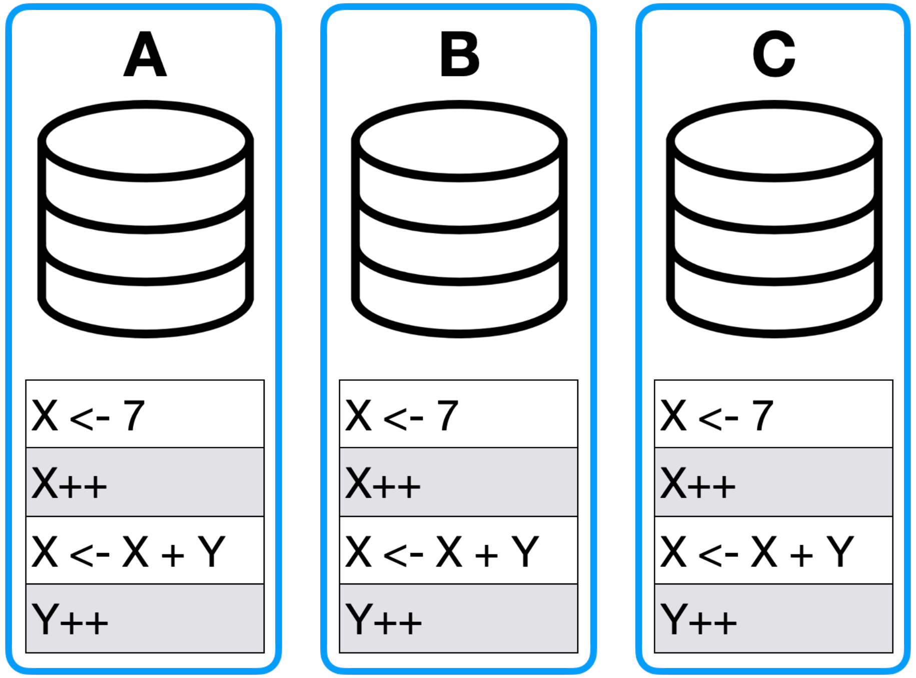
A recipe for a robust distributed service




Operating Systems

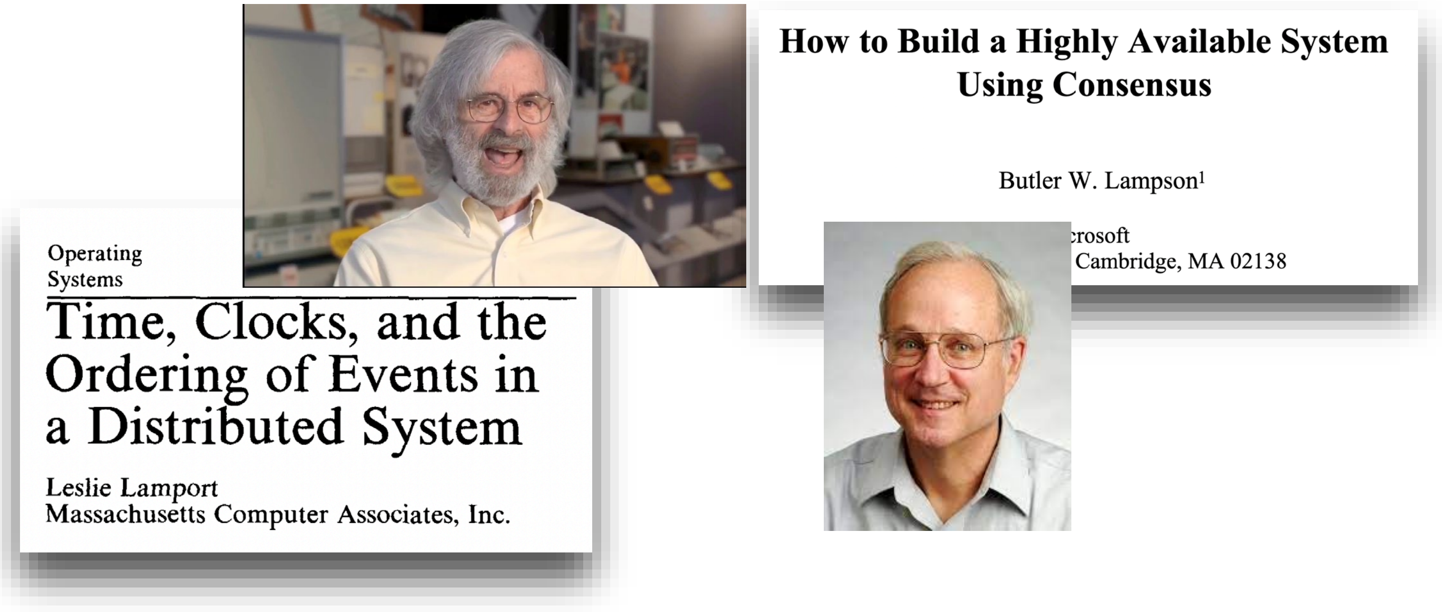
Time, Clocks, and the Ordering of Events in a Distributed System

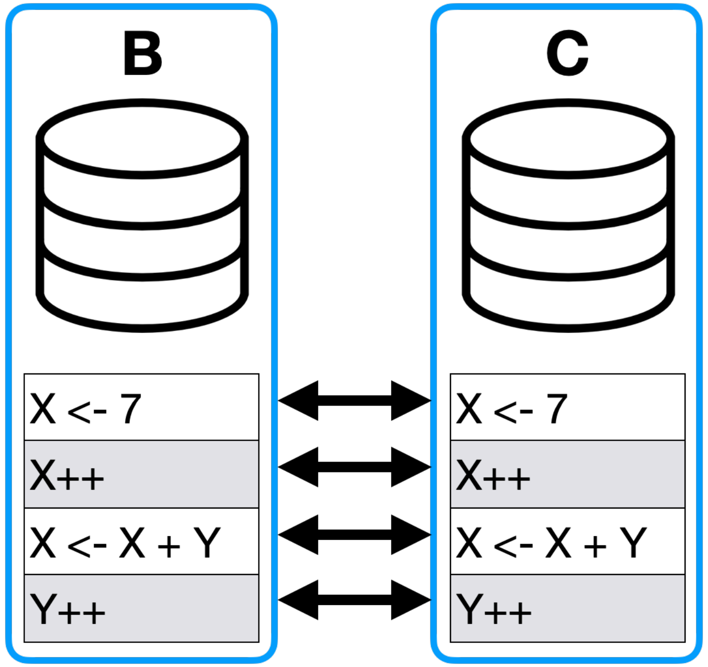
Leslie Lamport
Massachusetts Computer Associates, Inc.



1. Replicate your data
2. Make each replica execute an identical operation log

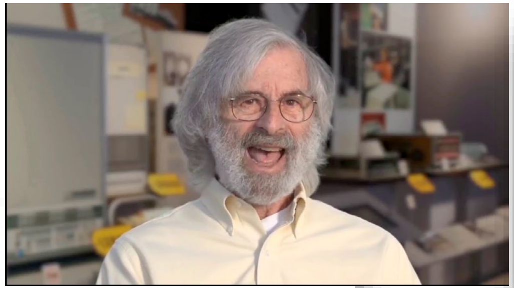
A recipe for a robust distributed service





1. Replicate your data
2. Make each replica execute an identical operation log
3. Use consensus protocol to agree on logs

A recipe for a robust distributed service



Operating Systems

Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport
Massachusetts Computer Associates, Inc.

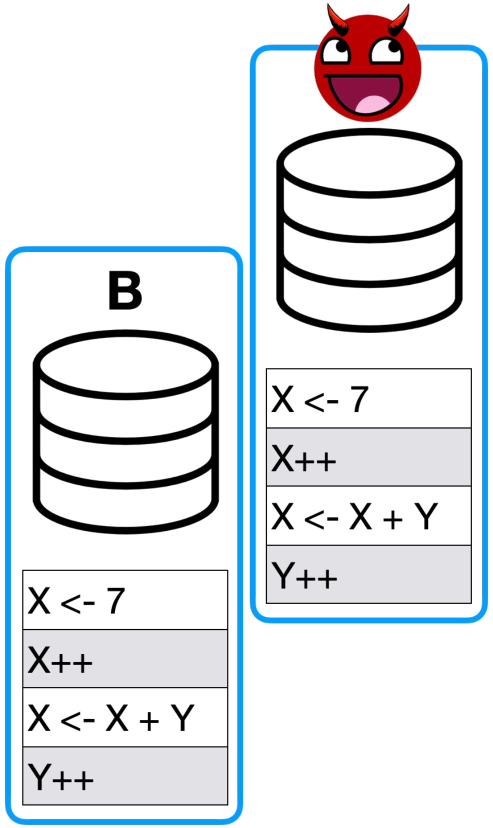
How to Build a Highly Available System Using Consensus

Butler W. Lampson[1]

crosoft
Cambridge, MA 02138

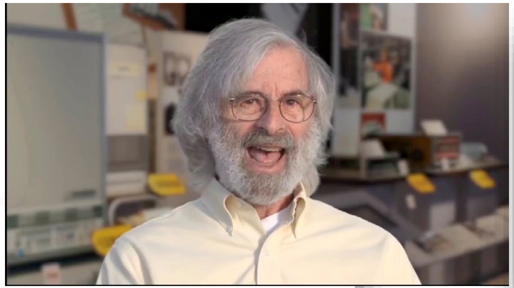
Practical Byzantine Fault Tolerance

Miguel Castro and Barbara Liskov
Laboratory for Computer Science,
Massachusetts Institute of Technology,
545 Technology Square, Cambridge, MA 02139
{castro,liskov}@lcs.mit.edu



1. Replicate your data
2. Make each replica execute an identical operation log
3. Use consensus protocol to agree on logs
4. Make it Byzantine fault-tolerant

A recipe for a robust distributed service



Operating Systems

Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport
Massachusetts Computer Associates, Inc.

How to Build a Highly Available System Using Consensus

Butler W. Lampson[1]
crosoft
Cambridge, MA 02138

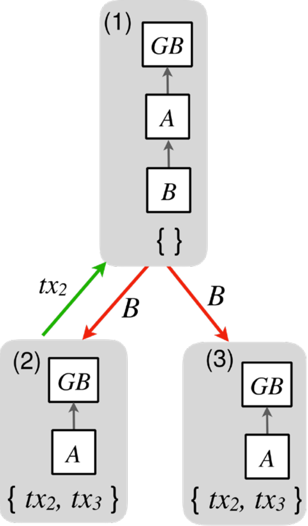
Practical Byzantine Fault Tolerance

Miguel Castro and Barbara Liskov
Laboratory for Computer Science,
Massachusetts Institute of Technology,
545 Technology Square, Cambridge, MA 02139
{castro,liskov}@lcs.mit.edu

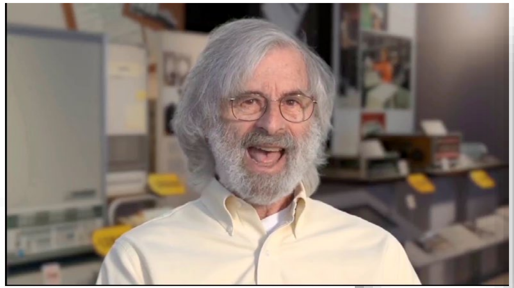
Bitcoin: A Peer-to-Peer Electronic Cash System

Satoshi Nakamoto
satoshin@gmx.com
www.bitcoin.org

1. Replicate your data
2. Make each replica execute an identical operation log
3. Use consensus protocol to agree on logs
4. Make it Byzantine fault-tolerant
5. Make it so anyone can participate



A recipe for a robust distributed service



Operating Systems

Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport
Massachusetts Computer Associates, Inc.

How to Build a Highly Available System Using Consensus

Butler W. Lampson[1]

crosoft
Cambridge, MA 02138

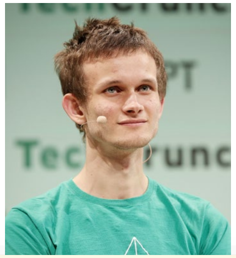
Practical Byzantine Fault Tolerance

Miguel Castro and Barbara Liskov
Laboratory for Computer Science,
Massachusetts Institute of Technology,
545 Technology Square, Cambridge, MA 02139
{castro,liskov}@lcs.mit.edu

Bitcoin: A Peer-to-Peer Electronic Cash System

Satoshi Nakamoto
satoshin@gmx.com
www.bitcoin.org

1. Replicate your data
2. Make each replica execute an identical operation log
3. Use consensus protocol to agree on logs
4. Make it Byzantine fault-tolerant
5. Make it so anyone can participate
6. Add arbitrary computations with costs



ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER
BERLIN VERSION 8fea825 – 2022-08-22

DR. GAVIN WOOD
FOUNDER, ETHEREUM & PARITY
GAVIN@PARITY.IO

What are the problems?

- **Correctness:**
  arbitrary replicated computations (aka *smart contracts*) can be *buggy* and allow for exploits

Examples:
- DAO reentrancy attack
- Parity Multi-Sig Wallet hack

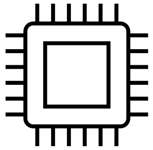
100s of papers published since 2016 on verification and vulnerability detection

- **Scalability:**
  consensus protocols are *slow* due to decentralization; no advantage from inherent distributed parallelism

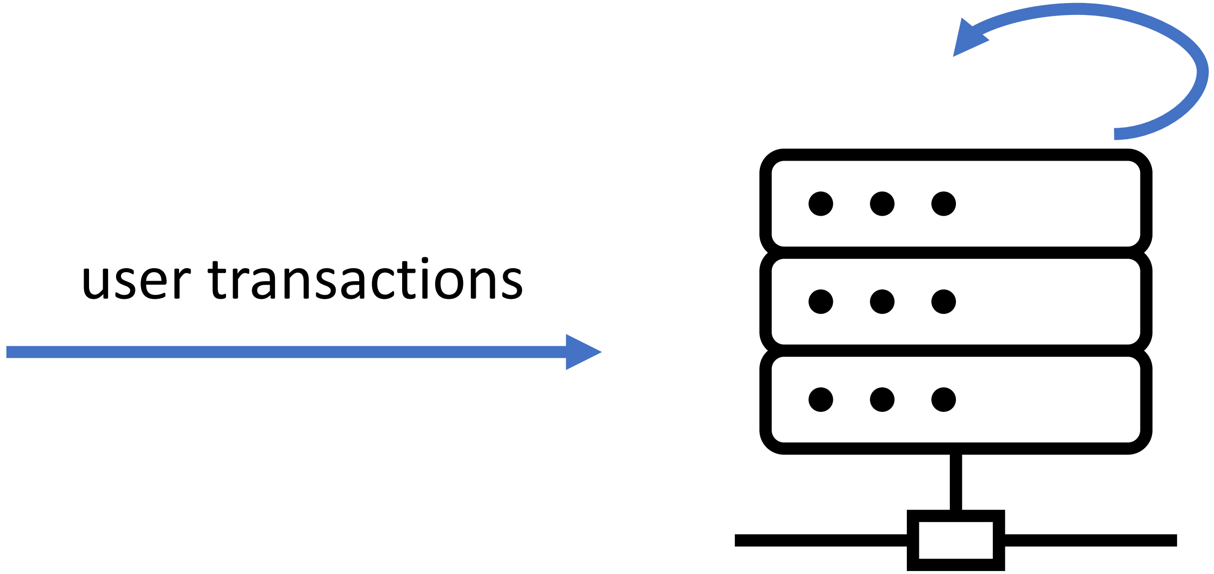
Examples:
- Ethereum only handles 11 TPS
- A single popular decentralised application (e.g., CryptoKitties) can cause congestion of the system

Scaling bottlenecks: state and execution

each node
executes every
transaction

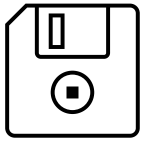


user transactions



network nodes

adding more nodes does not increase throughput OR allow more state to be stored



each node stores
the entire state

[1] https://twitter.com/peter_szilagyi/status/1460202014919569410

Scaling bottlenecks: state and execution

- Ethereum's head **state size** was ~130 GB as of Nov 2021[1]
  - For best performance, this needs to be kept in RAM
  - In practice, it is disk-based (NVME SSD) with caching in memory:
    - AFAIK, most of the time spent processing an Ethereum transaction is spent on disk I/O
- Increasing node hardware requirements decreases decentralisation
  - Solana validators already need >> 256 GB of RAM and 1 Gbps network
- In monolithic architectures, transaction **execution throughput** is limited by the capacity of the *least performant node* in the network

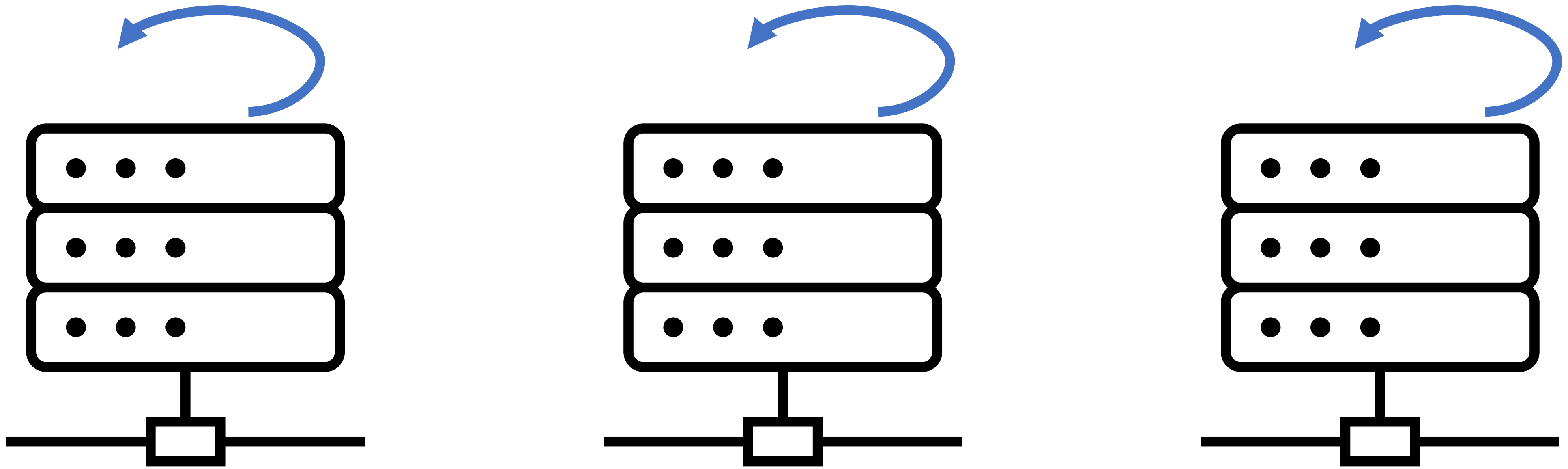
Scaling Blockchains: State of the Art

- **Layer 1** solutions
  Revise the *rules of the consensus* protocol to increase the throughput
  - Changing Proof-of-Work to *Proof-of-Stake*
  - Parallelism via *sharding*

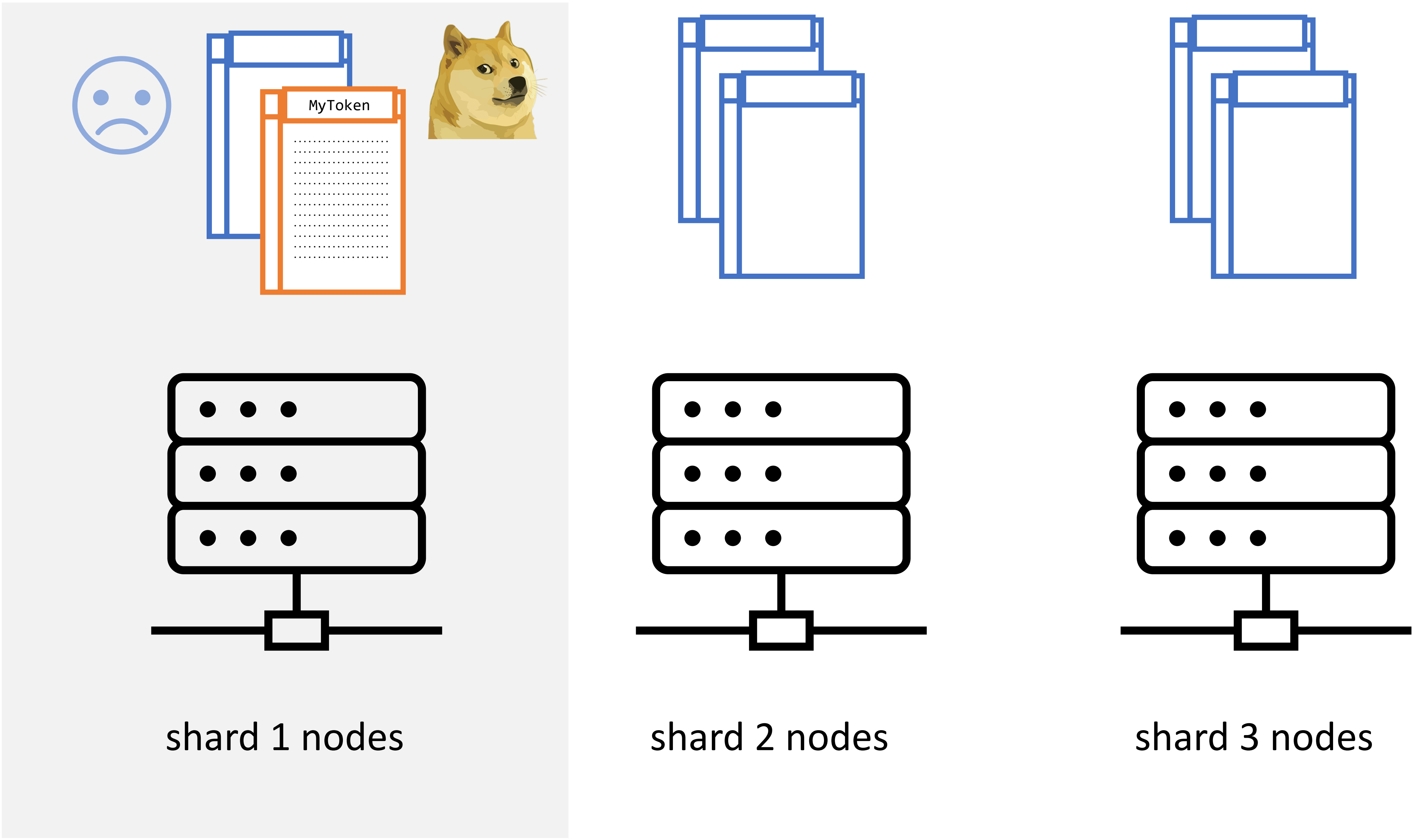
- **Layer 2** solutions
  Adding *auxiliary protocols* to offload transaction processing
  - *Nested blockchains*: the main chain stores results of side-chain transactions
  - *Off-chain executions* via zkSNARKs and optimistic roll-ups

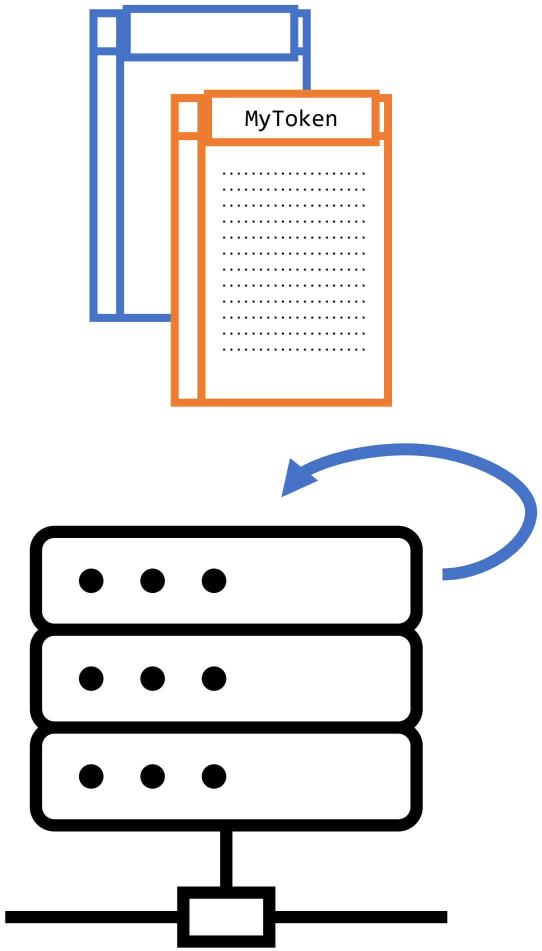
Sharding

each shard
executes a subset
of transactions

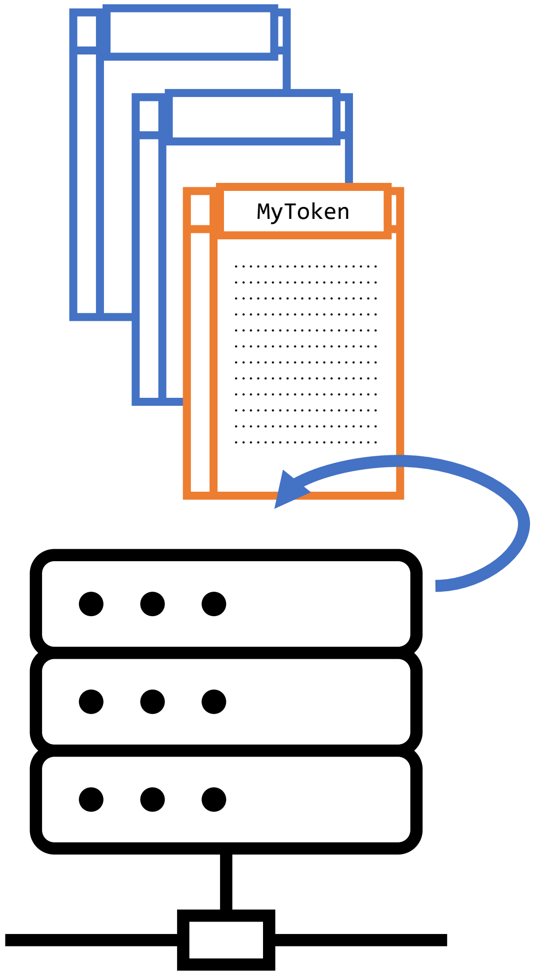


shard 1 nodes shard 2 nodes shard 3 nodes

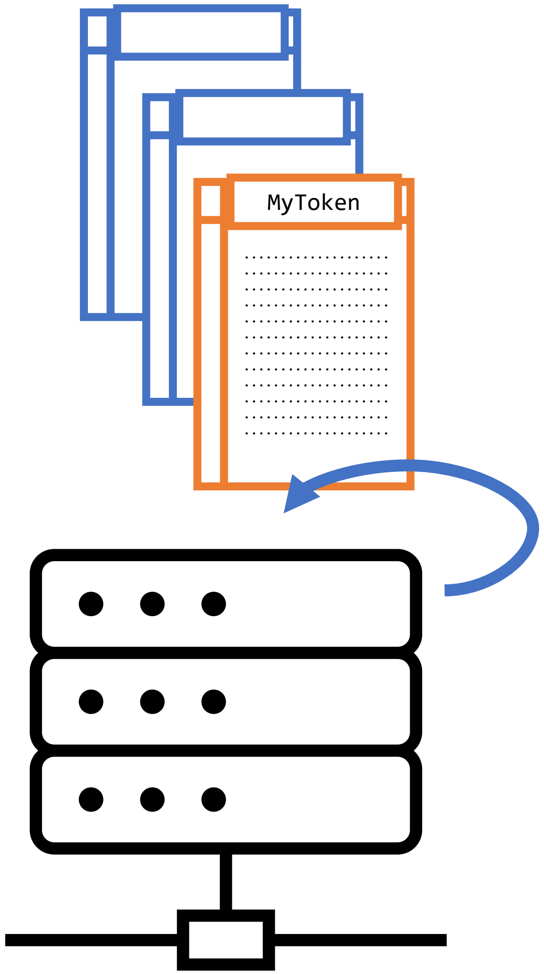

MyToken
shard 1 nodes
shard 2 nodes
shard 3 nodes



shard 1 nodes



shard 2 nodes



shard 3 nodes

Look at the contract's code to learn how to shard it!

problem we are working on

Smart Contract Sharding

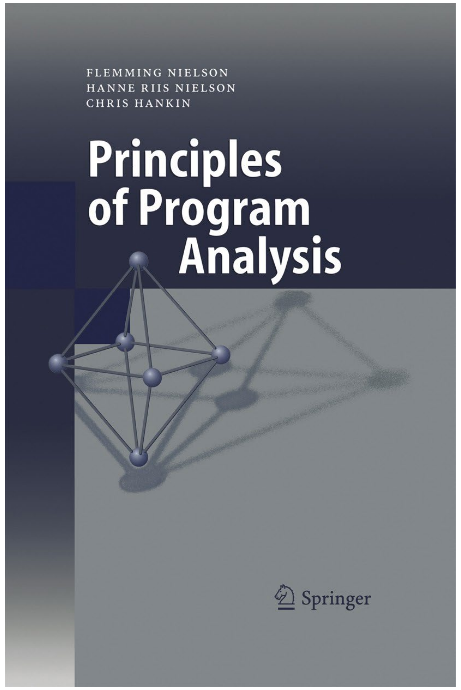
with Static Program Analysis

Ownership

Commutativity

technique we are applying

Goals of Static Program Analysis



- Verification and Validation
  - Without running a program, soundly *prove the absence of bugs*…
    (e.g., Astrée Static Analyzer)
  - …or soundly *show their presence*
    (e.g., Coverity, FindBugs, Infer/RacerD/Pulse(X))
- Uncovering opportunities for program optimization
  - Constant propagation, function inlining, static method dispatch
    (e.g., any optimizing compiler)
  - Automated parallelization

Smart Contract *Sharding* with **Static Program Analysis**

2

≈

Formal reasoning

1

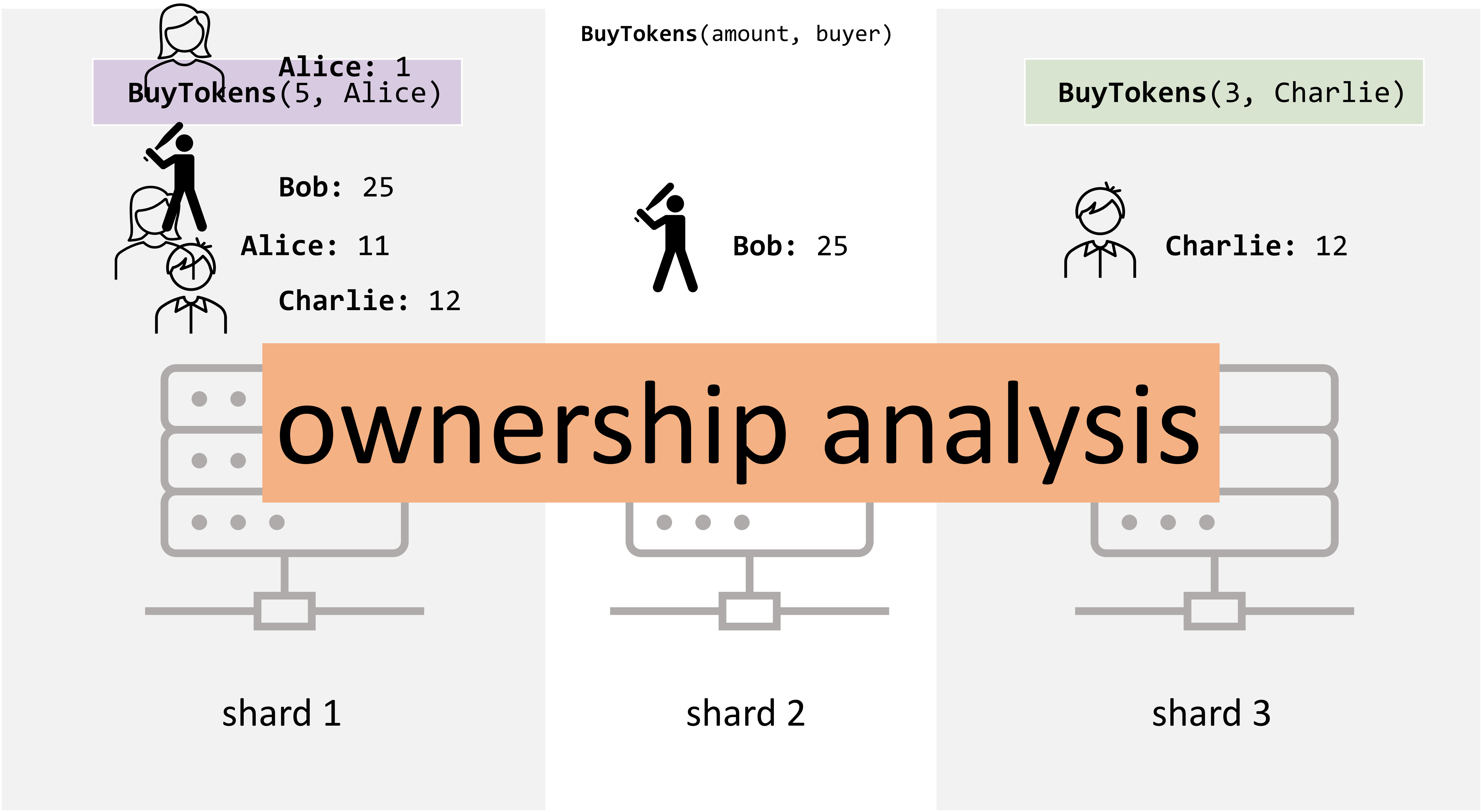
for *scalability*,

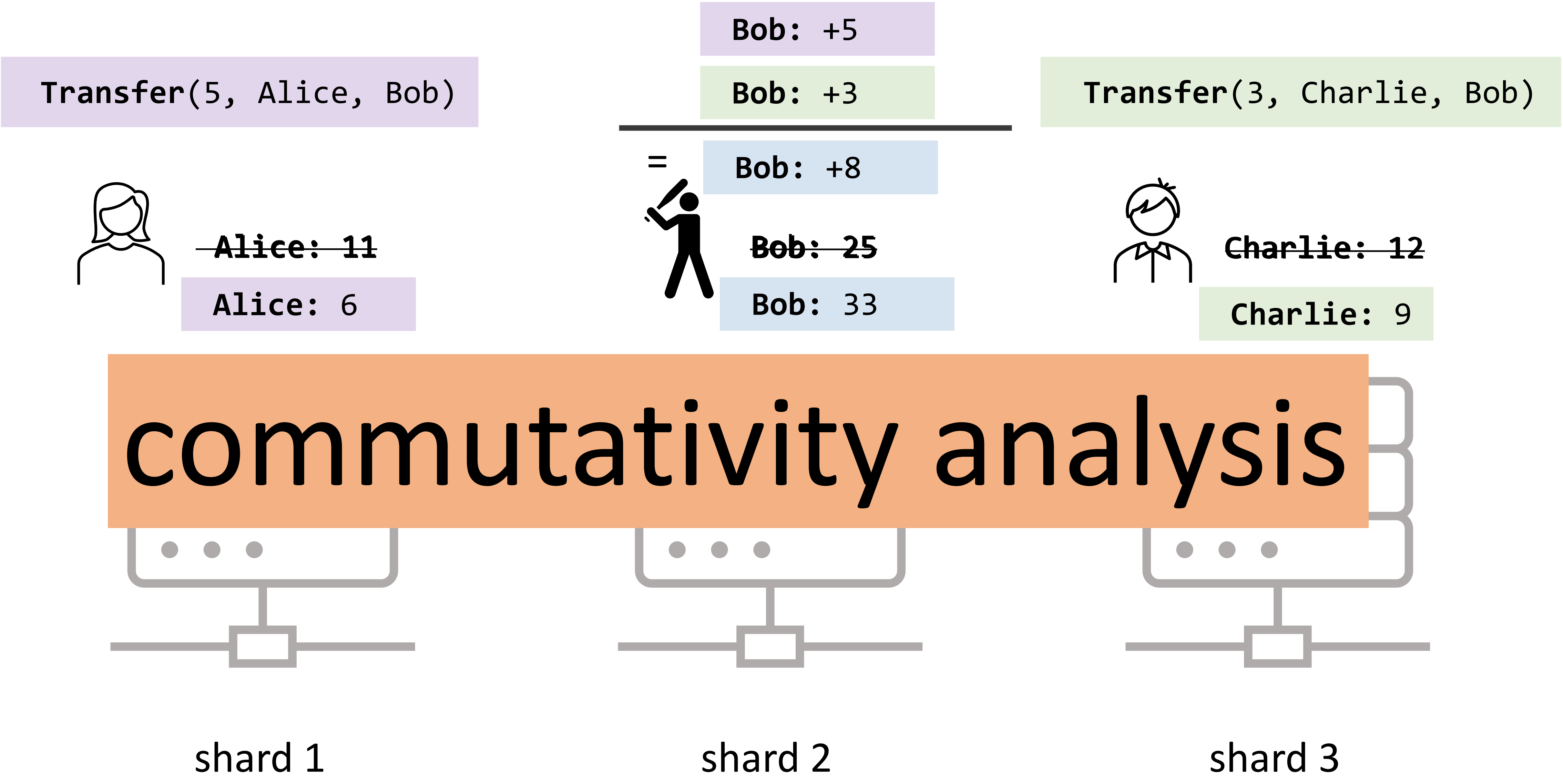
not just verification

MyToken

```
Balances:
    Alice:   10
    Bob:     25
    Charlie: 12

Transitions:
  BuyTokens(amount, buyer)
  Transfer(amount, from, to)
```


BuyTokens(amount, buyer)
Alice: 1
BuyTokens(5, Alice)
Bob: 25
Alice: 11
Charlie: 12
Bob: 25
BuyTokens(3, Charlie)
Charlie: 12
ownership analysis
shard 1
shard 2
shard 3


Transfer(5, Alice, Bob)
Bob: +5
Bob: +3
= Bob: +8
Transfer(3, Charlie, Bob)
~~Alice: 11~~
Alice: 6
~~Bob: 25~~
Bob: 33
~~Charlie: 12~~
Charlie: 9
commutativity analysis
shard 1
shard 2
shard 3

CoSplit

Static Program Analysis
for
Smart Contract Sharding

ZILLIQA

+

SCILLA

SCILLA

OOPSLA'19

Artifacts Available

Artifacts Evaluated Reusable

Artifacts Evaluated Functional

Safer Smart Contract Programming with Scilla

ILYA SERGEY, Yale-NUS College, Singapore and National University of Singapore, Singapore
VAIVASWATHA NAGARAJ, Zilliqa Research, India
JACOB JOHANNSEN, Zilliqa Research, Denmark
AMRIT KUMAR, Zilliqa Research, United Kingdom
ANTON TRUNOV, Zilliqa Research, Russia
KEN CHAN GUAN HAO, Zilliqa Research, Malaysia

Two key features:

- Clearly **separates computation from communication**
  - message-passing rather than method calls for contract interaction
- Strict distinction between **pure and effectful computations**
  - Scilla has a *small imperative fragment* with conditionals but without loops
  - Only pure (non-effectful) recursion is allowed

static analysis can be quite precise

```
field balances: Map Address Uint
```

```
transition Transfer(to: Address, amount: Uint)
  from_bal <- balances[_sender];
  match from_bal with
  | Some bal =>
    match amount ≤ bal with
    | True =>
      new_from_bal = builtin sub bal amount;
      balances[_sender] := new_from_bal;
      to_bal <- balances[to];
      new_to_bal = match to_bal with
      | Some bal => builtin add bal amount
      | None => amount
      end;
      balances[to] := new_to_bal
```

Static analysis for transition effects

- *Ownership:* produce an **effect summary** for every transition
  - Effects include: reads, writes, accepting funds, sending messages, conditioning on values derived from mutable fields
  - The effect summary *over-approximates* the behaviour of the transition
  - Loosely inspired by Concurrent Separation Logic

- *Commutativity:* **linearity-aware flows-to analysis**
  - Effects of monotone operations, which use a field just once, *commute*
  - Inspired by GHC's cardinality analysis (POPL'14)
  - Expressed as a type system for "contribution types"
    - compositional, but sometimes gives uninformative types

Constant	x, y	constant contract field or transition parameter
Mutable	f	mutable field or map-field access via parameter
Contrib. src.	$cs ::=$	$x \mid \mathsf{f}$
Cardinality	$card ::=$	$\mathsf{None} \mid \mathsf{Linear} \mid \mathsf{NonLinear}$
Operation	$op ::=$	$+ \mid - \mid * \mid \ldots$
Abstr. expr.	$e ::=$	$\top \mid \overline{(cs, card, \overline{op})}$
Effect	$\varepsilon ::=$	$\mathsf{Read}(\mathsf{f}) \mid \mathsf{Write}(\mathsf{f}, e) \mid \mathsf{AcceptFunds} \mid$
		$\mathsf{Condition}(e) \mid \mathsf{Event}(e) \mid \mathsf{SendMsg}(e) \mid \top$

```
transition Transfer(to: Address, amount: Uint)
  from_bal <- balances[_sender];
  match from_bal with
  | Some bal =>
    match amount ≤ bal with
    | True =>
      new_from_bal = builtin sub bal amount;
      balances[_sender] := new_from_bal;
      to_bal <- balances[to];
      new_to_bal = match to_bal with
      | Some bal => builtin add bal amount
      | None => amount
      end;
      balances[to] := new_to_bal
```

Read(balances[_sender])

Condition(balances[_sender])

(balances[_sender], Linear, Ø)

Condition(balances[_sender], amount)

{(balances[_sender], Linear, sub),
(amount, Linear, sub)}

Write(balances[_sender],
{(balances[_sender], Linear, sub),
(amount, Linear, sub)})

Read(balances[to])

Write(balances[to],
{(balances[to], Linear, add),
(amount, Linear, add)})

$$\text{Constraint} \qquad oc ::= \mathsf{Owns}(\mathsf{f}) \mid \mathsf{UserAddr}(x) \mid \mathsf{NoAliases}(\langle x, y \rangle) \mid$$
$$\mathsf{SenderShard} \mid \mathsf{ContractShard} \mid \bot$$

Constraint $oc ::= \mathsf{Owns}(\mathsf{f}) \mid \mathsf{UserAddr}(x) \mid \mathsf{NoAliases}(\langle x, y \rangle) \mid \mathsf{SenderShard} \mid \mathsf{ContractShard} \mid \bot$

Join $\uplus_{\mathsf{f}} ::= \mathsf{OwnOverwrite} \mid \mathsf{IntMerge}$

Weak reads

Owns(balances[_sender])

NoAliases(<_sender, to>)

OwnOverwrite join for owned contributions
IntMerge join for un-owned contributions

Read(balances[_sender])

Condition(balances[_sender])

Condition(balances[_sender], amount)

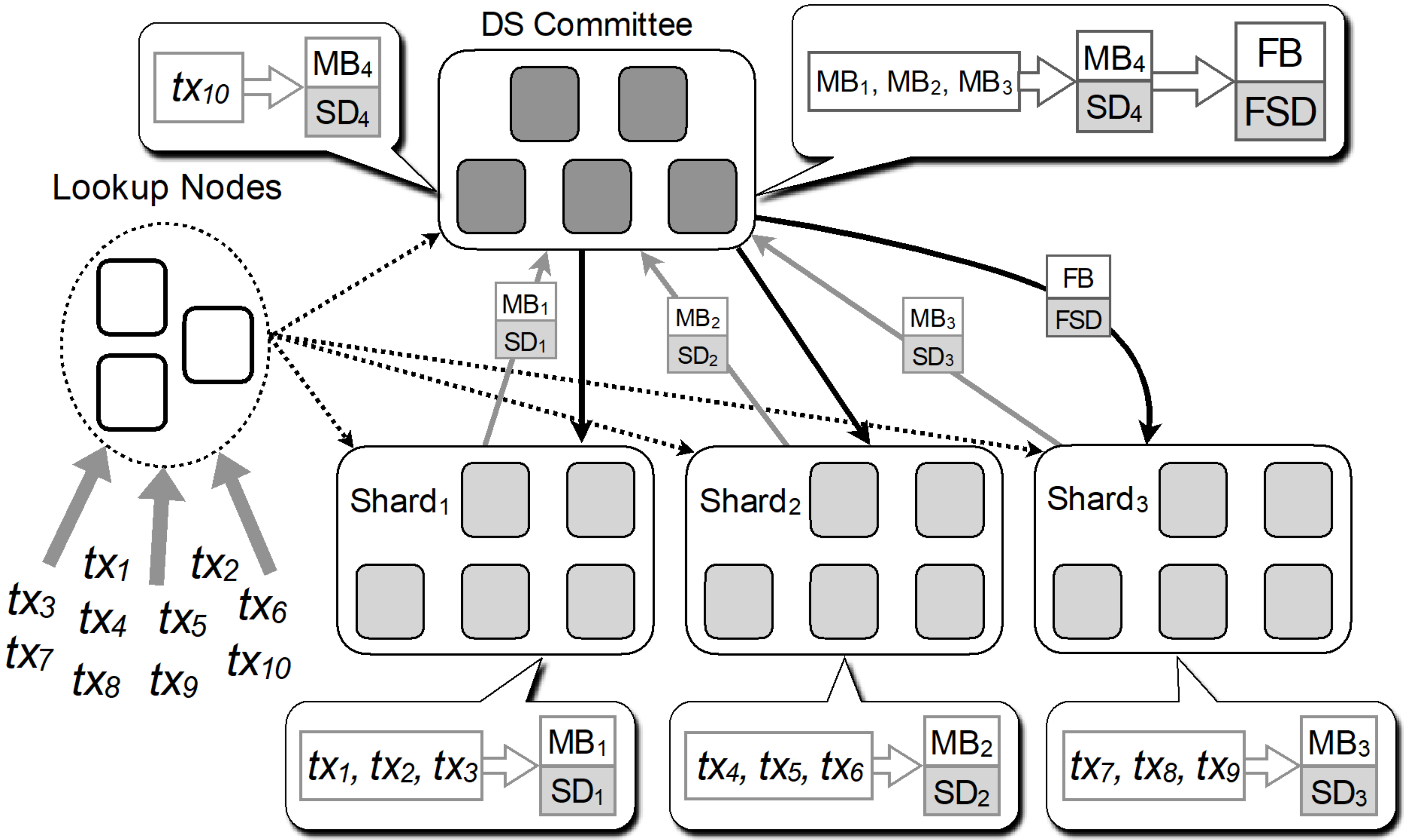
Write(balances[_sender],
{(balances[_sender], Linear, sub),
(amount, Linear, sub)})

Read(balances[to])

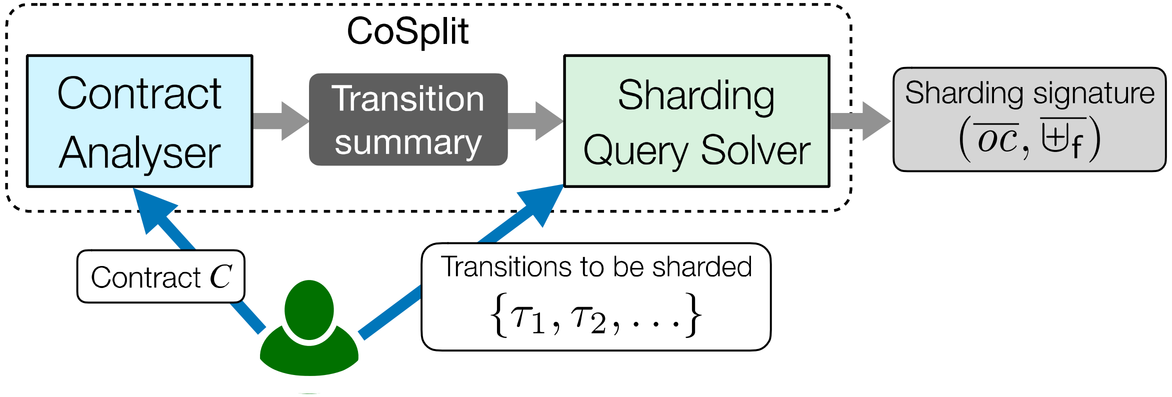
Write(balances[to],
{(balances[to], Linear, add),
(amount, Linear, add)})

Integration

A sharded blockchain design



Integrating CoSplit with Zilliqa



1. Run the static analysis when the contract is first deployed
2. Store the resulting *sharding signature*

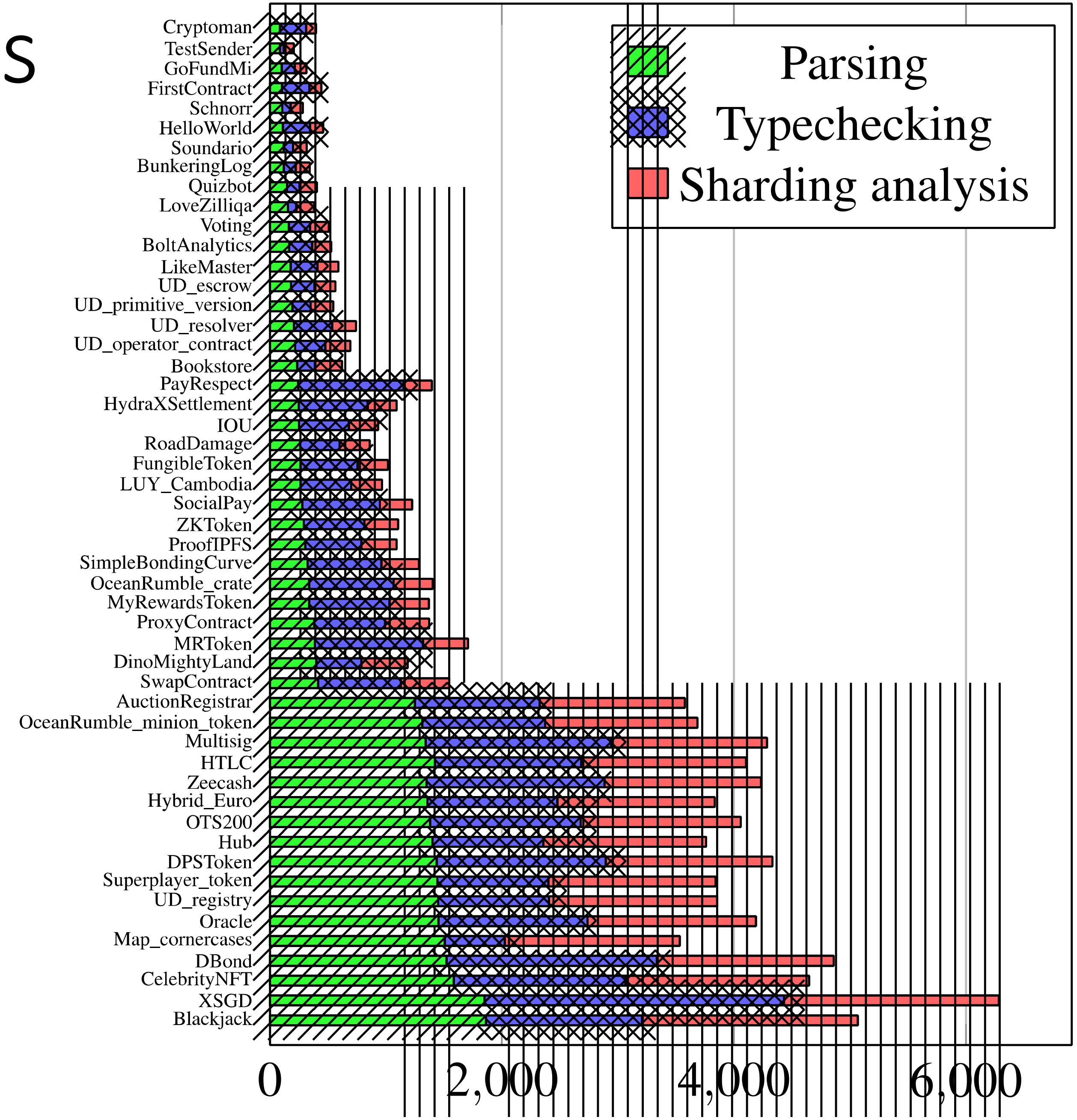
   = set of transition constraints + join instructions for each field in the contract

3. When processing a transaction, *solve the constraints* to determine which shard(s) the transaction can be processed by
   - if the constraints have no solution, must process sequentially/cross-shard

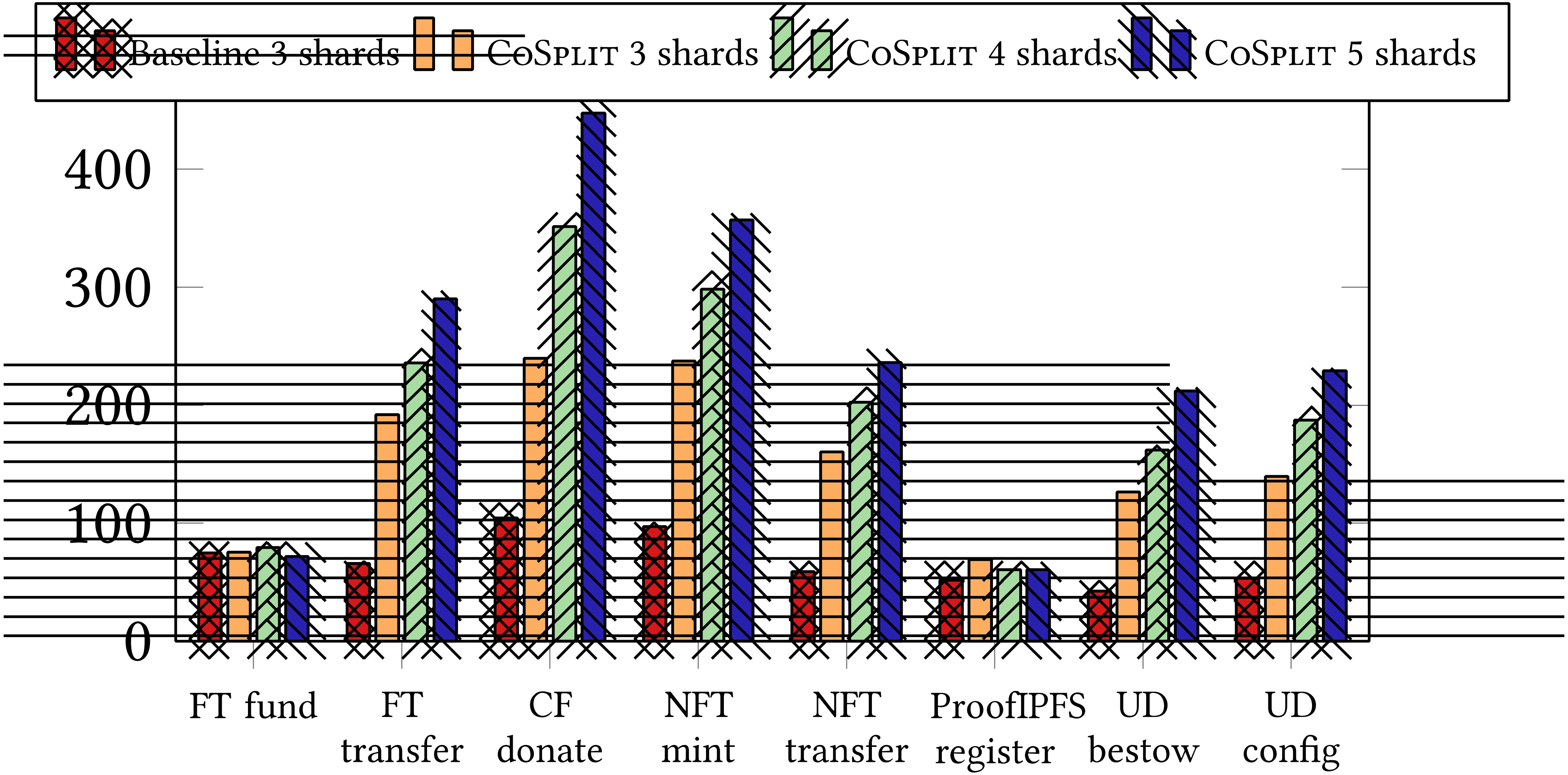
4. After parallel processing, *merge (join) state contributions* from shards before sequential transactions are processed

Evaluation

Static Overheads



Throughput: Transactions per Second



Limitations and Discussion

- Currently no support for sharding multi-contract transactions
  - We would need to somehow combine the signatures from multiple contracts
- Some contracts require simple rewriting to be shardable
  - An opportunity for program repair

```
transition transfer(to: ByStr20, tokenId: Uint256)
  getTokenOwner <- tokenOwners[tokenId];
  match getTokenOwner with
  | None => throw
  | Some tokenOwner =>
    isOwner = builtin eq _sender tokenOwner;
    (* … *)
    getOperatorStatus <-
      operatorApprovals[tokenOwner][_sender];
    (* … *)
    tokenOwners[tokenId] := to;
```

```
transition transfer(tokenOwner: ByStr20,
                    to: ByStr20, tokenId: Uint256)
  getTokenOwner <- tokenOwners[tokenId];
  match getTokenOwner with
  | None => throw
  | Some actual =>
    isCorrectOwner = builtin eq tokenOwner actual;
    match isCorrectOwner with
    | False => throw
    | True =>
      isOwner = builtin eq _sender tokenOwner;
      (* … *)
      getOperatorStatus <-
        operatorApprovals[tokenOwner][_sender];
      (* … *)
      tokenOwners[tokenId] := to;
```

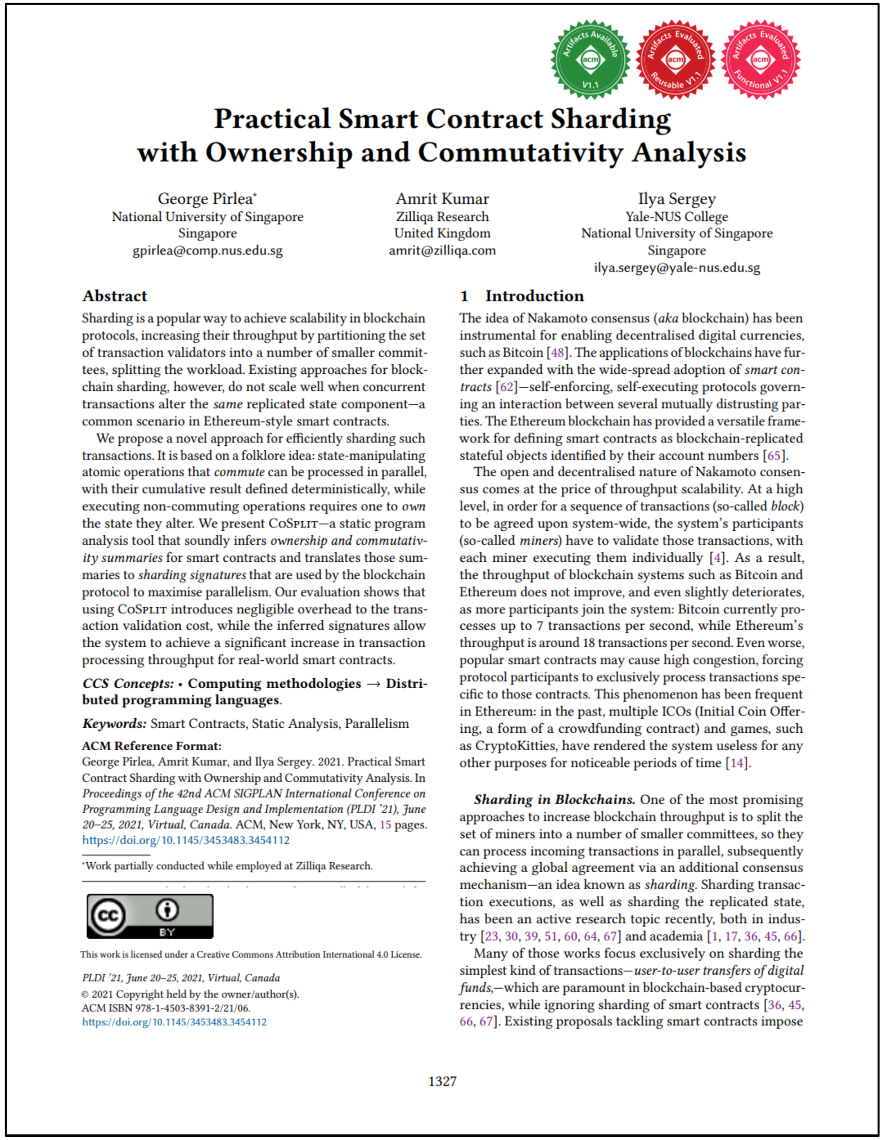
Limitations and Discussion

- Currently no support for sharding multi-contract transactions
  - We would need to somehow combine the signatures from multiple contracts
- Some contracts require simple rewriting to be shardable
  - An opportunity for program repair
- Some programming languages (e.g., Move or Solana's Rust dialect) might be even better targets for sharding analysis
  - one can also ask the programmer for ownership/commutativity annotations

Conclusion: What this talk was about

a parallelising compiler
for blockchains

To Take Away



Practical Smart Contract Sharding with Ownership and Commutativity Analysis

George Pîrlea*
National University of Singapore
Singapore
gpirlea@comp.nus.edu.sg

Amrit Kumar
Zilliqa Research
United Kingdom
amrit@zilliqa.com

Ilya Sergey
Yale-NUS College
National University of Singapore
Singapore
ilya.sergey@yale-nus.edu.sg

Abstract

Sharding is a popular way to achieve scalability in blockchain protocols, increasing their throughput by partitioning the set of transaction validators into a number of smaller committees, splitting the workload. Existing approaches for blockchain sharding, however, do not scale well when concurrent transactions alter the *same* replicated state component—a common scenario in Ethereum-style smart contracts.

We propose a novel approach for efficiently sharding such transactions. It is based on a folklore idea: state-manipulating atomic operations that *commute* can be processed in parallel, with their cumulative result defined deterministically, while executing non-commuting operations requires one to *own* the state they alter. We present CoSplit—a static program analysis tool that soundly infers *ownership and commutativity summaries* for smart contracts and translates those summaries to *sharding signatures* that are used by the blockchain protocol to maximise parallelism. Our evaluation shows that using CoSplit introduces negligible overhead to the transaction validation cost, while the inferred signatures allow the system to achieve a significant increase in transaction processing throughput for real-world smart contracts.

CCS Concepts: • **Computing methodologies → Distributed programming languages.**

Keywords: Smart Contracts, Static Analysis, Parallelism

ACM Reference Format:
George Pîrlea, Amrit Kumar, and Ilya Sergey. 2021. Practical Smart Contract Sharding with Ownership and Commutativity Analysis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3453483.3454112

*Work partially conducted while employed at Zilliqa Research.

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06.
https://doi.org/10.1145/3453483.3454112

1 Introduction

The idea of Nakamoto consensus (*aka* blockchain) has been instrumental for enabling decentralised digital currencies, such as Bitcoin [48]. The applications of blockchains have further expanded with the wide-spread adoption of *smart contracts* [62]—self-enforcing, self-executing protocols governing an interaction between several mutually distrusting parties. The Ethereum blockchain has provided a versatile framework for defining smart contracts as blockchain-replicated stateful objects identified by their account numbers [65].

The open and decentralised nature of Nakamoto consensus comes at the price of throughput scalability. At a high level, in order for a sequence of transactions (so-called *block*) to be agreed upon system-wide, the system's participants (so-called *miners*) have to validate those transactions, with each miner executing them individually [4]. As a result, the throughput of blockchain systems such as Bitcoin and Ethereum does not improve, and even slightly deteriorates, as more participants join the system: Bitcoin currently processes up to 7 transactions per second, while Ethereum's throughput is around 18 transactions per second. Even worse, popular smart contracts may cause high congestion, forcing protocol participants to exclusively process transactions specific to those contracts. This phenomenon has been frequent in Ethereum: in the past, multiple ICOs (Initial Coin Offering, a form of a crowdfunding contract) and games, such as CryptoKitties, have rendered the system useless for any other purposes for noticeable periods of time [14].

Sharding in Blockchains. One of the most promising approaches to increase blockchain throughput is to split the set of miners into a number of smaller committees, so they can process incoming transactions in parallel, subsequently achieving a global agreement via an additional consensus mechanism—an idea known as *sharding*. Sharding transaction executions, as well as sharding the replicated state, has been an active research topic recently, both in industry [23, 30, 39, 51, 60, 64, 67] and academia [1, 17, 36, 45, 66].

Many of those works focus exclusively on sharding the simplest kind of transactions—*user-to-user transfers of digital funds*,—which are paramount in blockchain-based cryptocurrencies, while ignoring sharding of smart contracts [36, 45, 66, 67]. Existing proposals tackling smart contracts impose

- ***Sharding*** is a solution to the blockchain scalability problem
- Some smart contract logic can be sharded (i.e., ***executed in parallel***)
- We developed ***static analysis*** to soundly determine sharding conditions for smart contracts
- The technique has been integrated into real-world blockchain and gave ***observable increase*** in the throughput

Thanks!