Scaling Decentralized Ledgers via Sharding

Ewa Syta

Trinity College

Swiss Blockchain Winter School

February 12, 2019

Talk Outline

Scalable Bias-Resistant Distributed Randomness

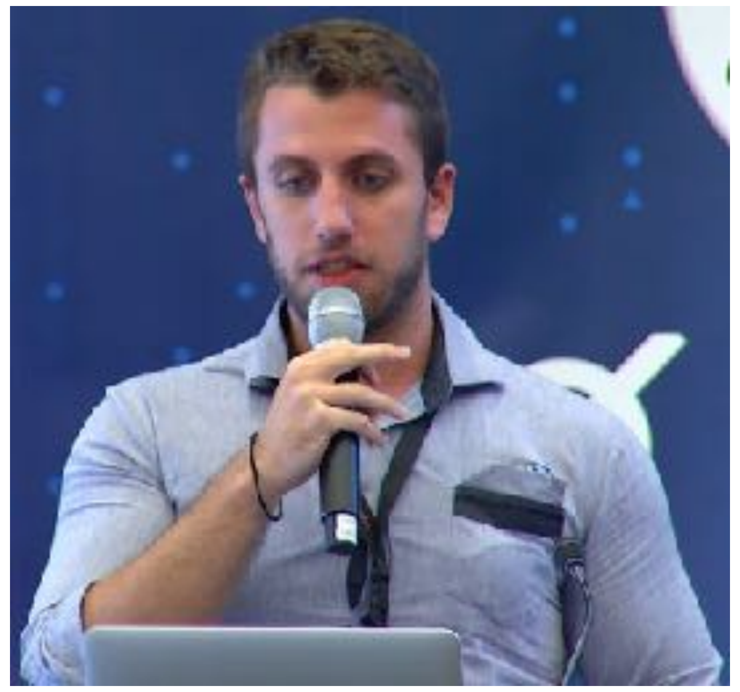
2017 IEEE Symposium on Security and Privacy

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

2018 IEEE Symposium on Security and Privacy

Acknowledgements


Philipp Jovanovic
(EPFL, CH)


Eleftherios Kokoris Kogias
(EPFL, CH)


Nicolas Gailly
(EPFL, CH)


Ismail Khoffi
(EPFL, CH)


Linus Gasser
(EPFL, CH)


Michael Fischer
(Yale University, USA)


Bryan Ford
(EPFL, CH)

Talk Outline

Scalable Bias-Resistant Distributed Randomness

2017 IEEE Symposium on Security and Privacy

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

2018 IEEE Symposium on Security and Privacy

Talk Outline

- Motivation
- Two Randomness Protocols
  - RandHound
  - RandHerd
- Implementation, Experimental Results and Current Deployment
- Conclusions

Talk Outline

- **Motivation**
- Two Randomness Protocols
  - RandHound
  - RandHerd
- Implementation, Experimental Results and Current Deployment
- Conclusions

Public Randomness

- **Different** from secret randomness
  - Secret randomness used for cryptographic keys, for example
- **Collectively** used
- Unpredictable ahead of time
- Not secret past a certain point in time
- Entropy is not enough

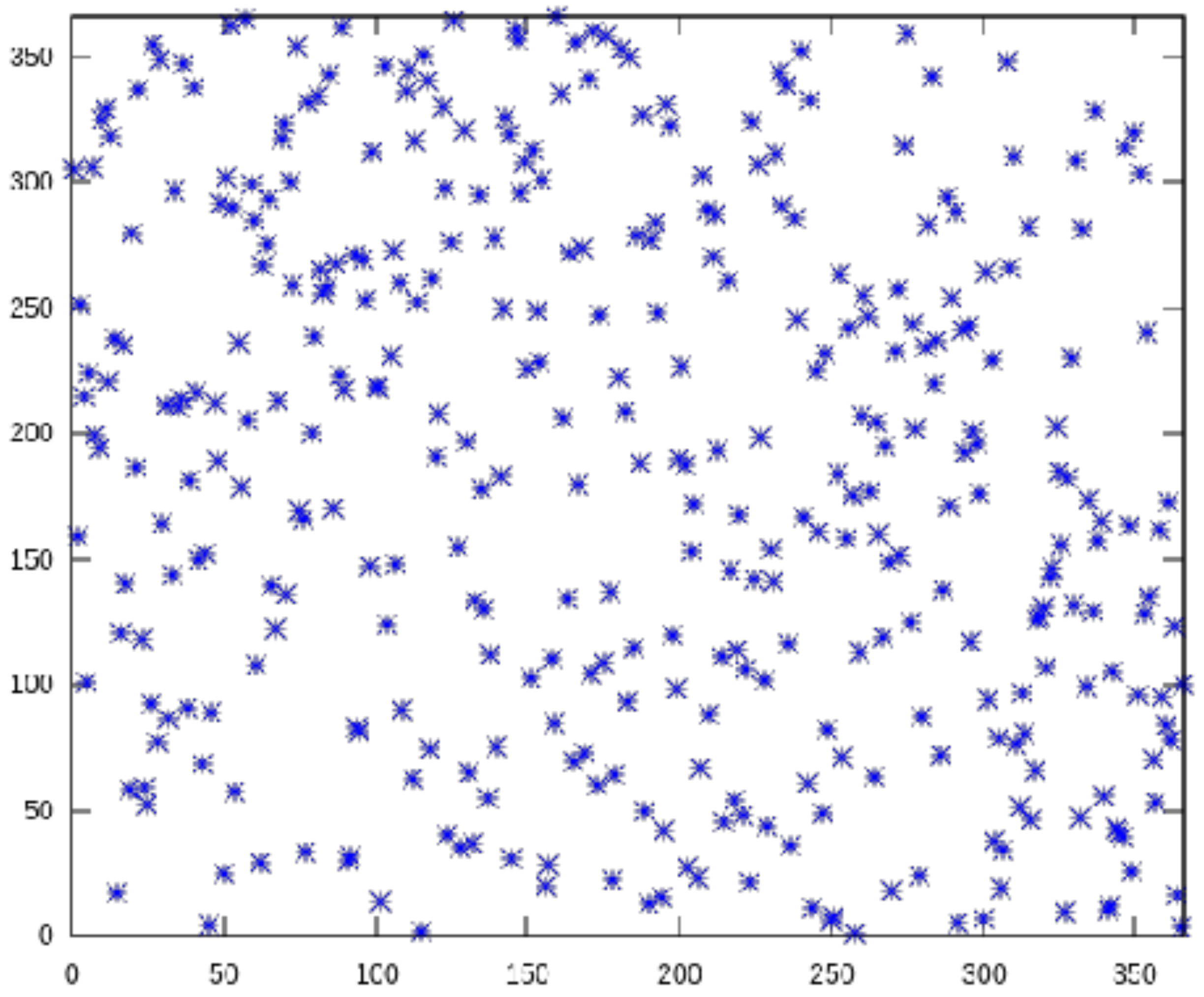
Applications of Public Randomness

- **Random selection**
  - lotteries, sweepstakes, jury selection, voting and election audit
- **Games**
  - shuffled decks, team assignments

- **Crypto**
  - challenges, authentication, cut-and-choose methods, “nothing up my sleeves” numbers
- **Protocols**
  - leader election for consensus protocols (PoS), sharding (OmniLedger), Tor (path selection)

Failed / Rigged Randomness

Vietnam War Lotteries (1969)



'European draws have been rigged': Ex-FIFA president Sepp Blatter claims to have seen hot and cold balls used to aid cheats

© EPA

Former FIFA president Sepp Blatter said he had witnessed rigged draws for European football competitions

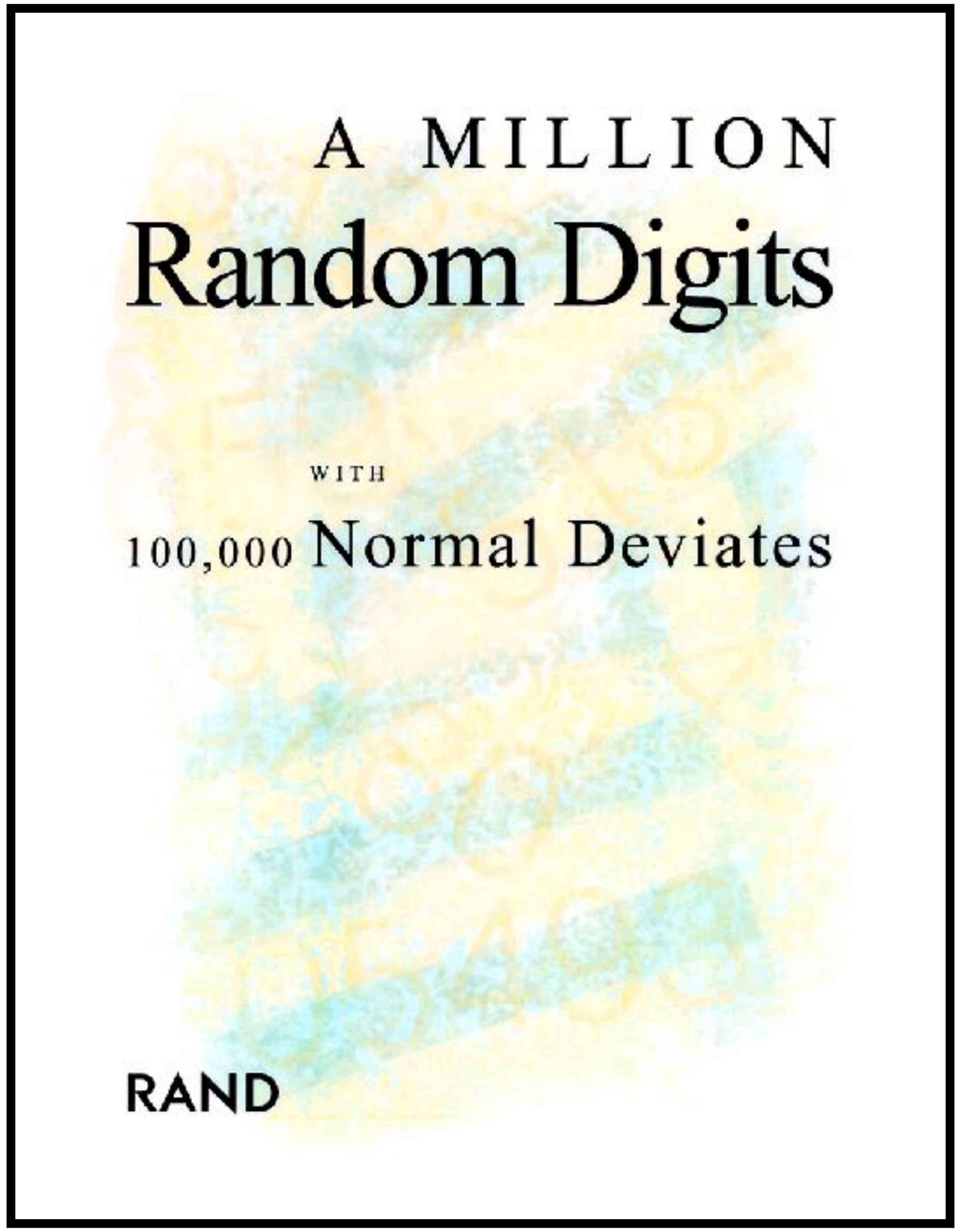
Man hacked random-number generator to rig lotteries, investigators say

New evidence shows lottery machines were rigged to produce predictable jackpot numbers on specific days of the year netting millions in winnings

'Computer whiz' rigged lottery number generator to produce predictable numbers a couple of times a year. Photograph: Brian Powers/AP

Public Randomness is not New

- 1955: Large table of random numbers published as a book by the Rand Corporation
- Today: Generating public random numbers is (still) **hard**
- Main issues: **trust** and **scale**
  - Both, in generation and usage



Goals

Decentralized, public randomness in the *(t,n)*-threshold security model

1. Availability

Successful protocol termination for up to *f=t-1* malicious nodes.

2. Unpredictability

Output not revealed prematurely.

3. Unbiasability

Output distributed uniformly at random.

4. Verifiability

Output correctness can be checked by third parties.

5. Scalability

Executable with hundreds of participants.

Assumptions: n= 3f +1, Byzantine adversary and asynchronous network with eventual message delivery

Public Randomness Approaches

- **With** Trusted Third Party
  - NIST Randomness Beacon

- **Without** TTP

Unusual assumptions

  - Bitcoin (Bonneau, 2015)
  - Slow cryptographic hash functions (Lenstra, 2015)
  - Lotteries (Baigneres, 2015)
  - Financial data (Clark, 2010)

(t,n)-threshold security model but not scalable

  - Coin-flipping (Cachin, 2015)
  - Distributed key generation (Kate, 2009)

Public Randomness is Hard

	Availability	Unpredictability	Unbiasability	Verifiability	Scalability
Strawman I	⛔	⛔	⛔	⛔	⛔
Strawman II	⛔	✅	⛔	⛔	⛔
Strawman III	✅	✅	☑️	⛔	⛔

Strawman I

- **Idea:** Combine random inputs of all participants.
- **Problem:** Last node fully controls output.

Strawman II

- **Idea:** Commit-then-reveal random inputs.
- **Problem:** Dishonest nodes can choose not to reveal.

Strawman III

- **Idea:** Secret-share random inputs.
- **Problem:** Dishonest nodes can send bad shares.

Public Randomness is Hard

	Availability	Unpredictability	Unbiasability	Verifiability	Scalability
Strawman I	⛔	⛔	⛔	⛔	⛔
Strawman II	⛔	✅	⛔	⛔	⛔
Strawman III	✅	✅	☑	⛔	⛔
RandShare	✅	✅	✅	⛔	⛔

RandShare

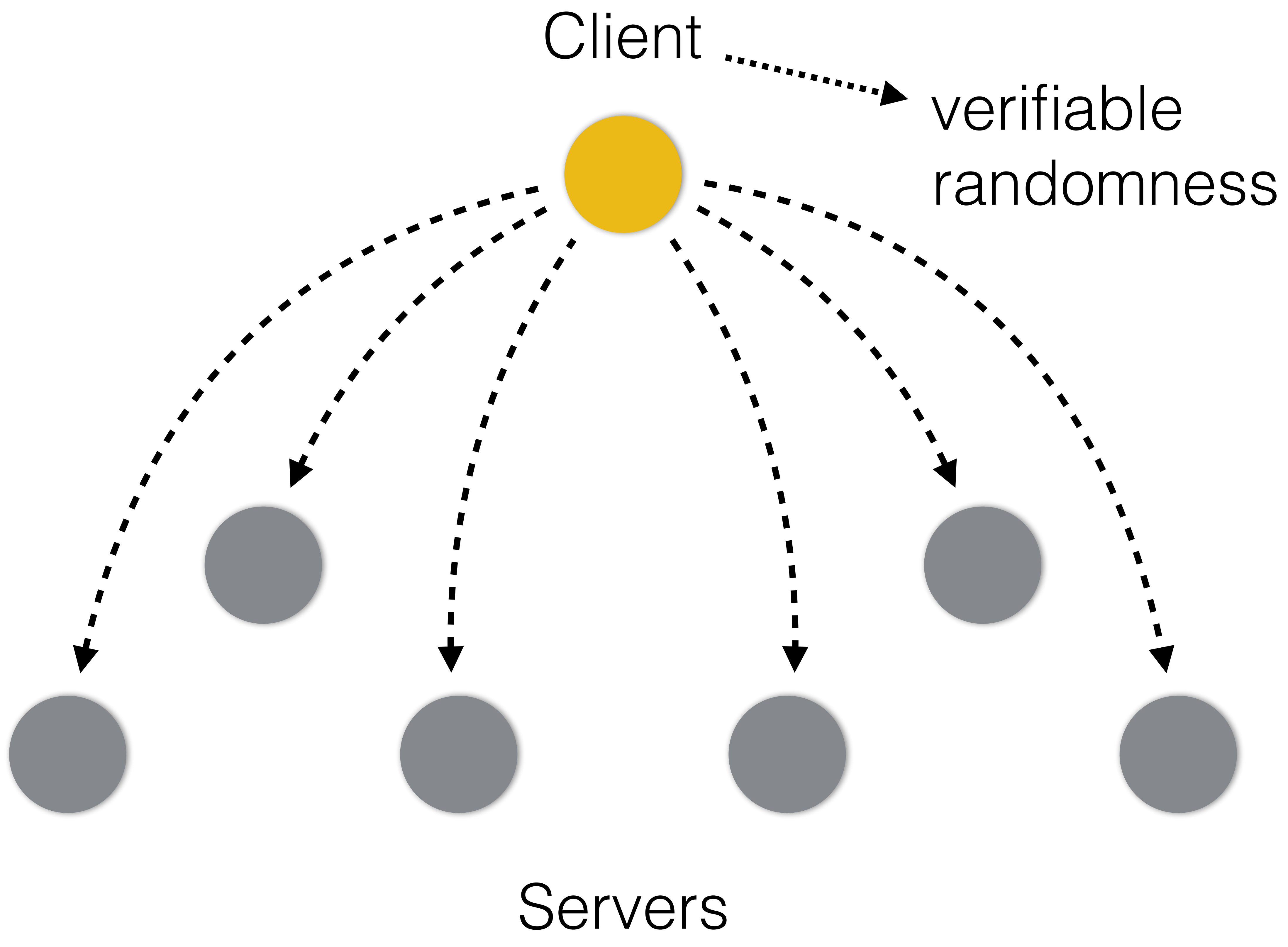
- **Idea:** Strawman III + *verifiable secret sharing* (Feldman, 1987)
- **Problems:**
  - Not publicly verifiable
  - Not scalable: $O(n^3)$ communication / computation complexity

Talk Outline

- Motivation
- **Two Randomness Protocols**
  - RandHound
  - RandHerd
- Implementation, Experimental Results and Current Deployment
- Conclusions

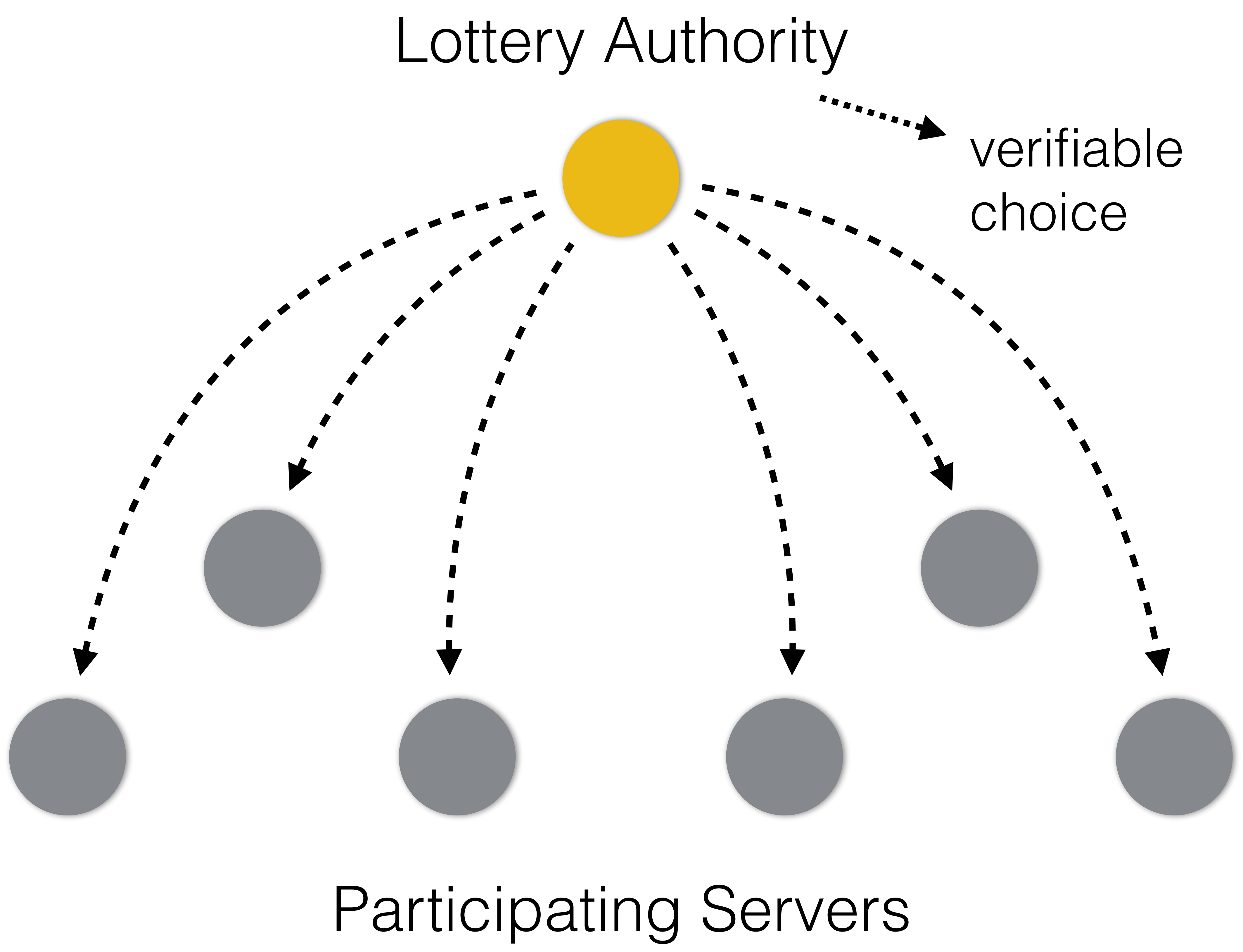
RandHound

- Goals
  - Verifiability: By third parties
  - Scalability: Performance better than $O(n^3)$
- Client/server randomness scavenging protocol
  - Untrusted client uses a large set of nearly-stateless servers
  - On demand (via configuration file)
  - One-shot approach



RandHound

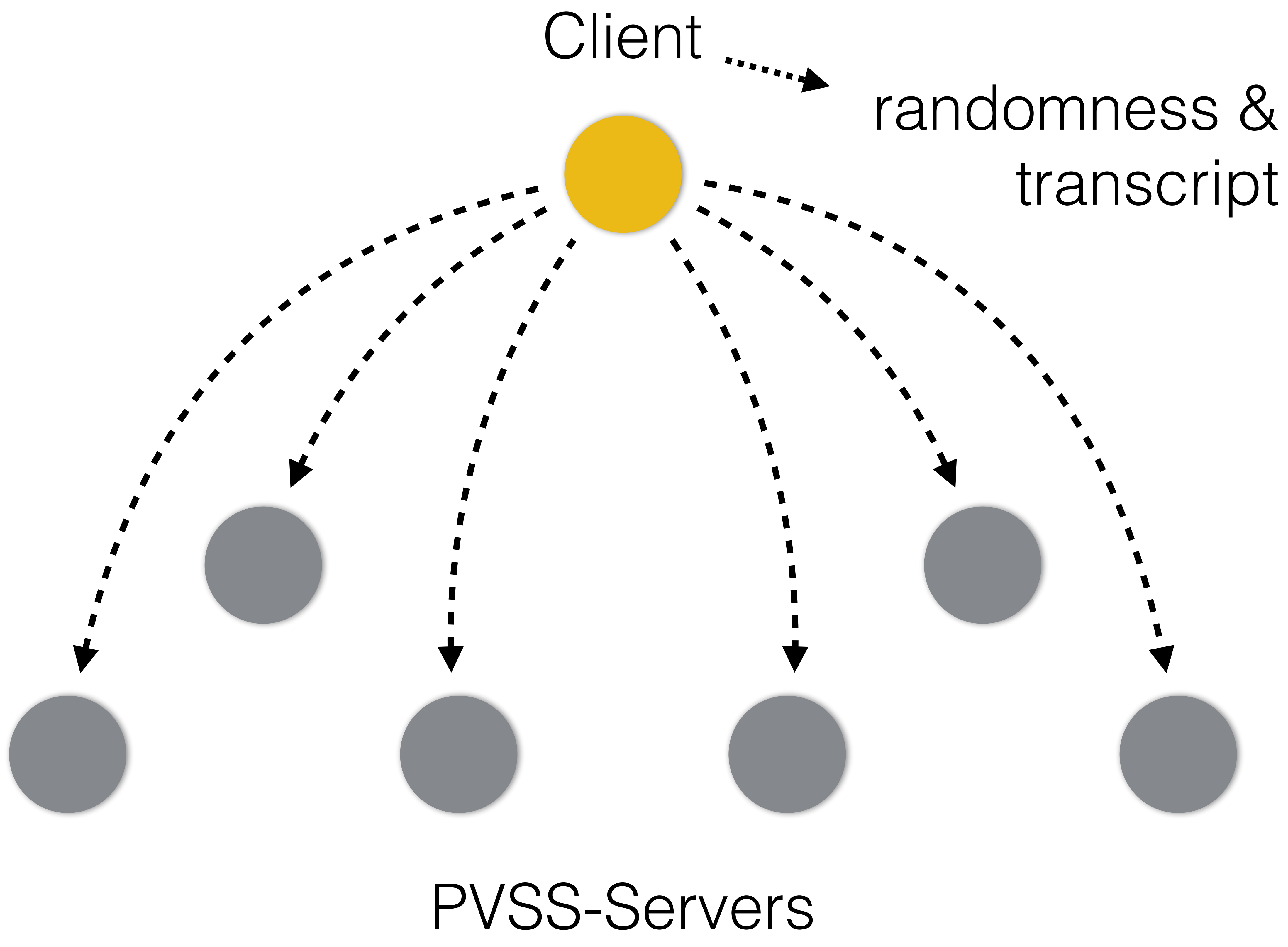
- Scenario
  - Lottery authority wants to pick a winner in a fair and verifiable process
- Setup
  - **Run**: announced in advance, publicly available config
  - **Client**: lottery authority
  - **Servers**: a set of reputable and independent parties
  - **Output**: randomness + third-party proof



RandHound

Achieving Public Verifiability

- Publicly-VSS (Schoenmakers, 1999)
  - Shares are encrypted and publicly verifiable through zero-knowledge proofs
  - No communication between servers
- CoSi Collective signing (Syta, 2016)
  - Client publicly commits to their choices
  - Any aggregate, threshold, or multi-signature
- Create protocol transcript from all sent/received messages



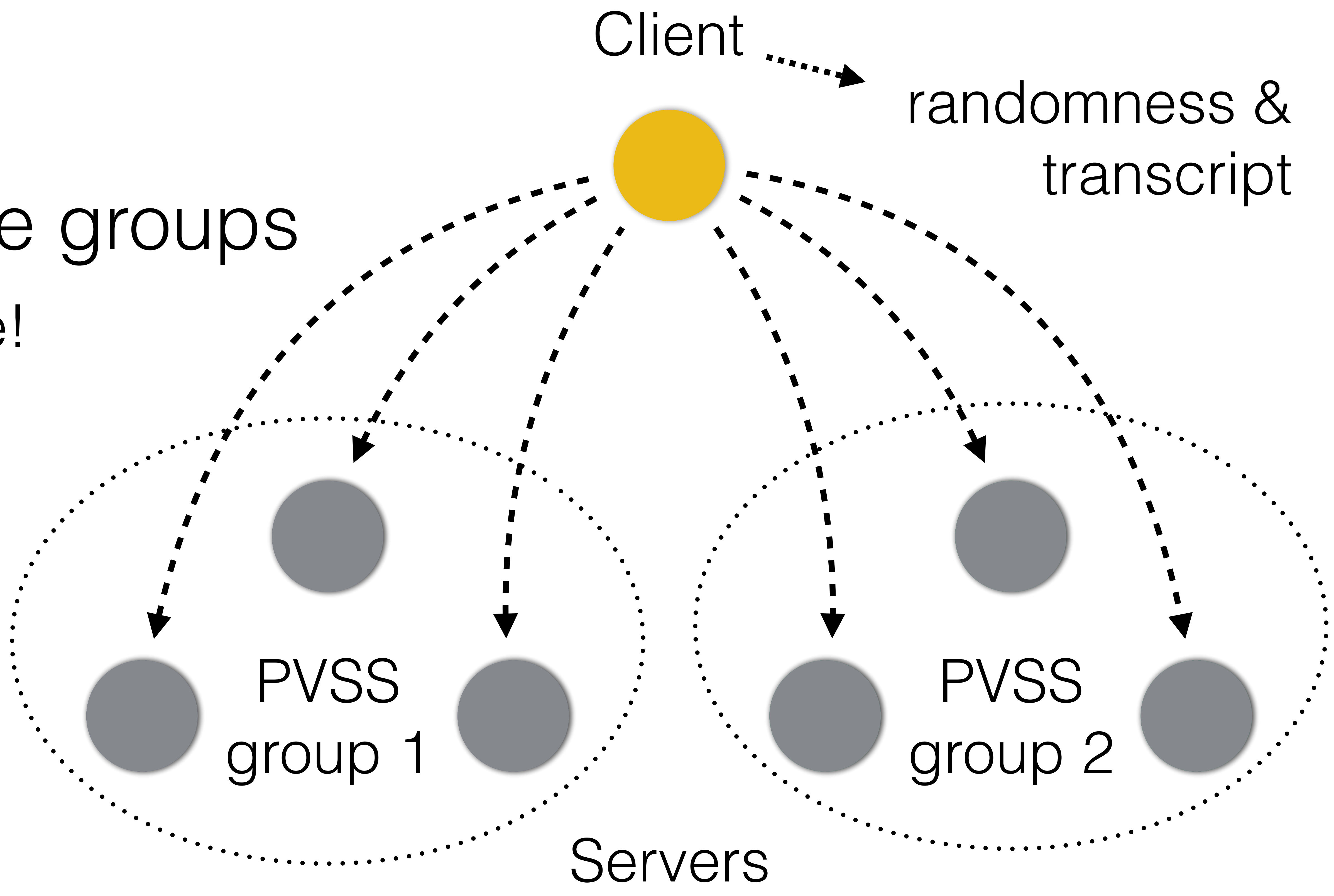
RandHound

Achieving Scalability

- Shard participants into constant size groups
  - Secret sharing with everyone too expensive!
  - Run secret sharing (only) inside groups
  - Collective randomness: combination of all group outputs

Chicken-and-Egg problem?

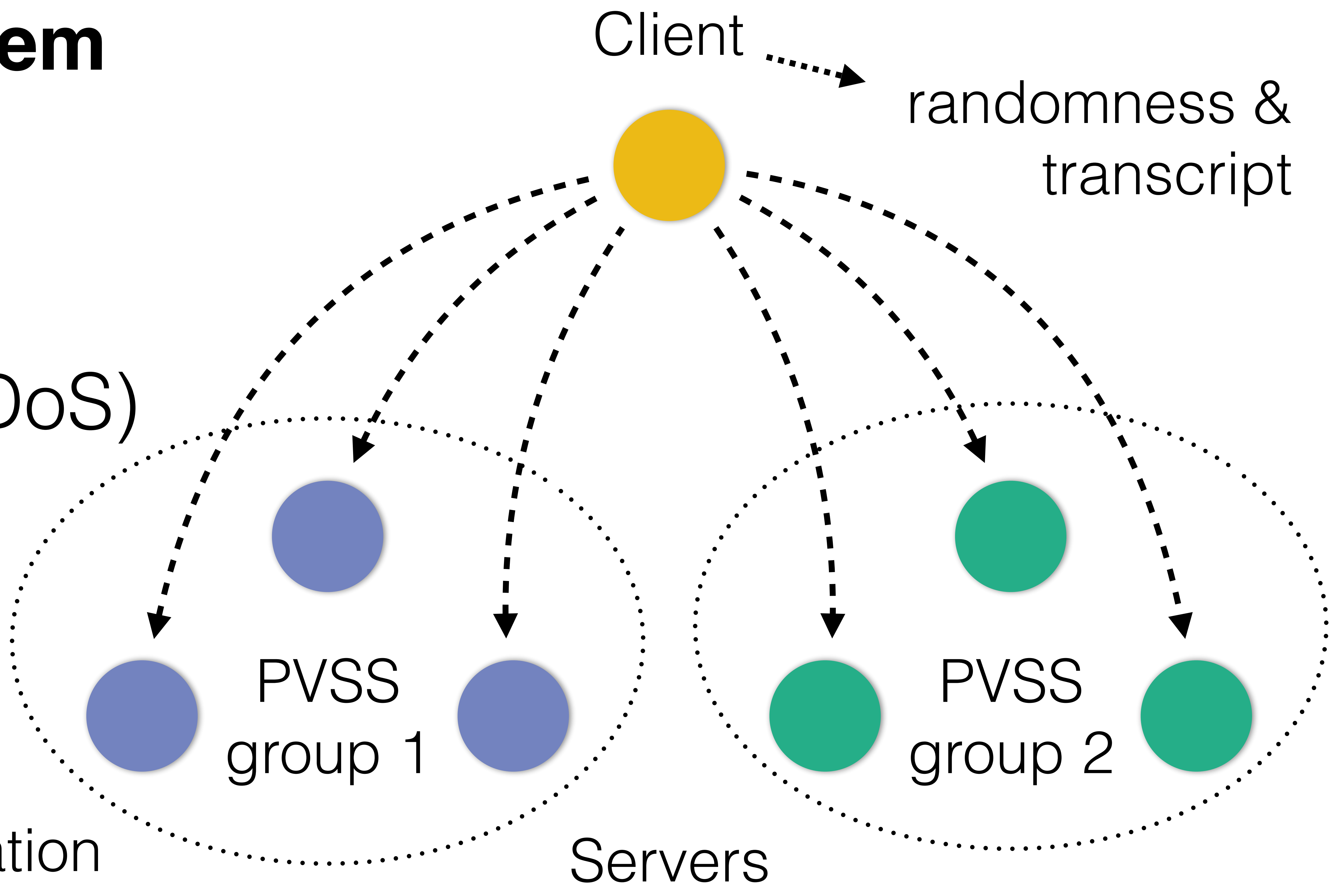
- How to securely assign participants to groups?



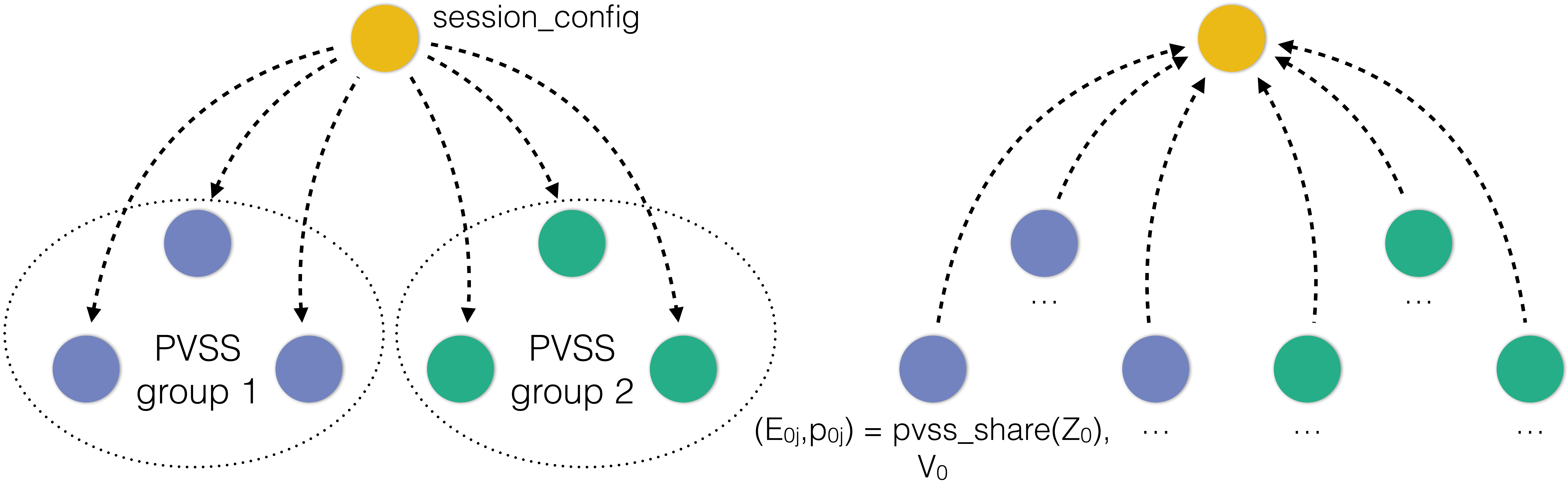
RandHound

Solving the Chicken-and-Egg Problem

- Client selects server grouping
- Availability might be affected (self-DoS)
- Security properties through
  - *Pigeonhole principle:* at least one group is not controlled by the adversary
  - *Collective signing:* prevents client equivocation by fixing the secrets that contribute to randomness



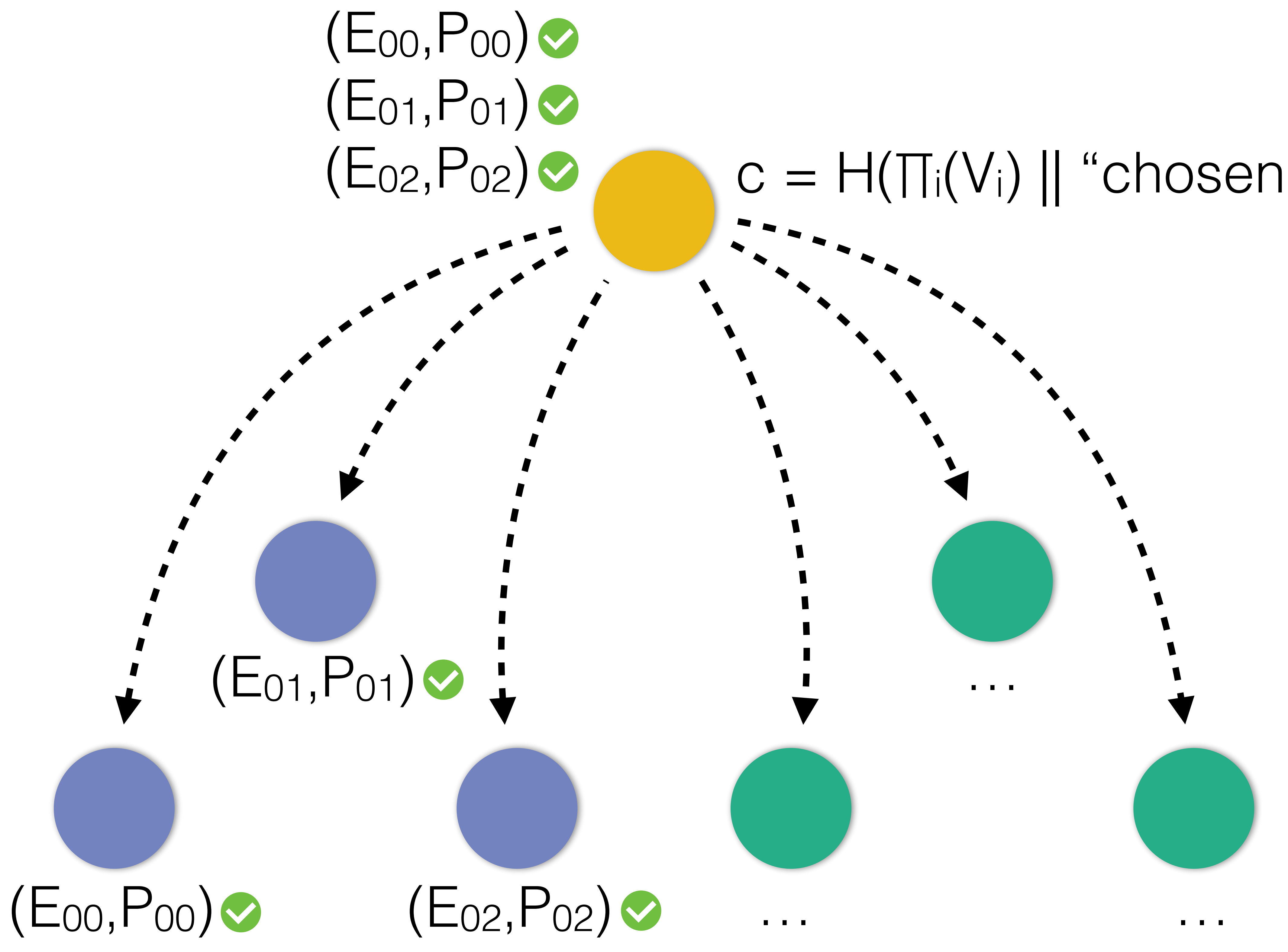
RandHound



1. Initialization (C)
Send session config,
divide servers into PVSS groups

2. Share Distribution (S)
Send encrypted PVSS shares,
CoSi commits

RandHound

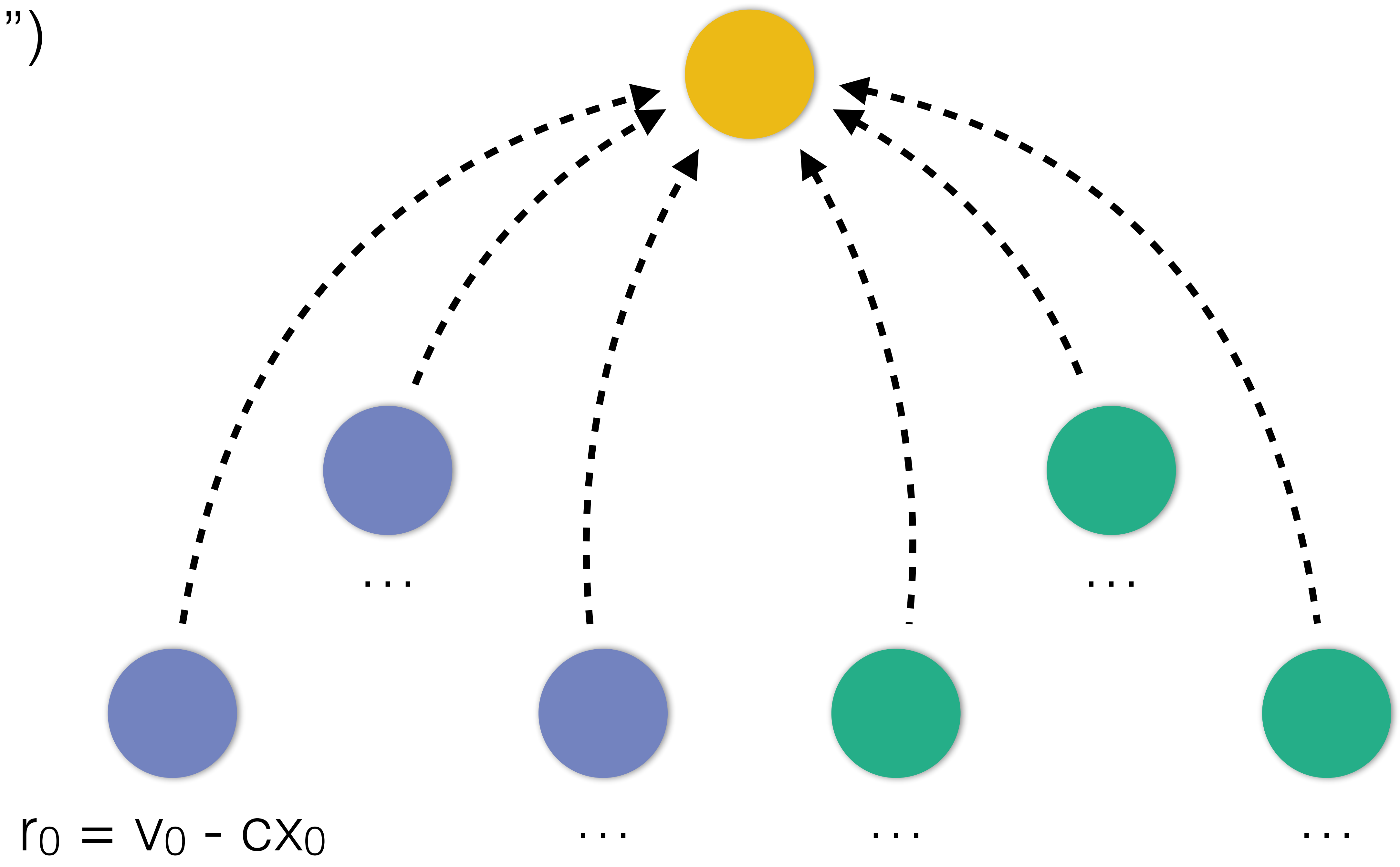


(E_{00},P_{00}) ✓
(E_{01},P_{01}) ✓
(E_{02},P_{02}) ✓

$c = H(\prod_i(V_i) \,||\, \text{“chosen secrets”})$

(E_{01},P_{01}) ✓ …

(E_{00},P_{00}) ✓ (E_{02},P_{02}) ✓ … …



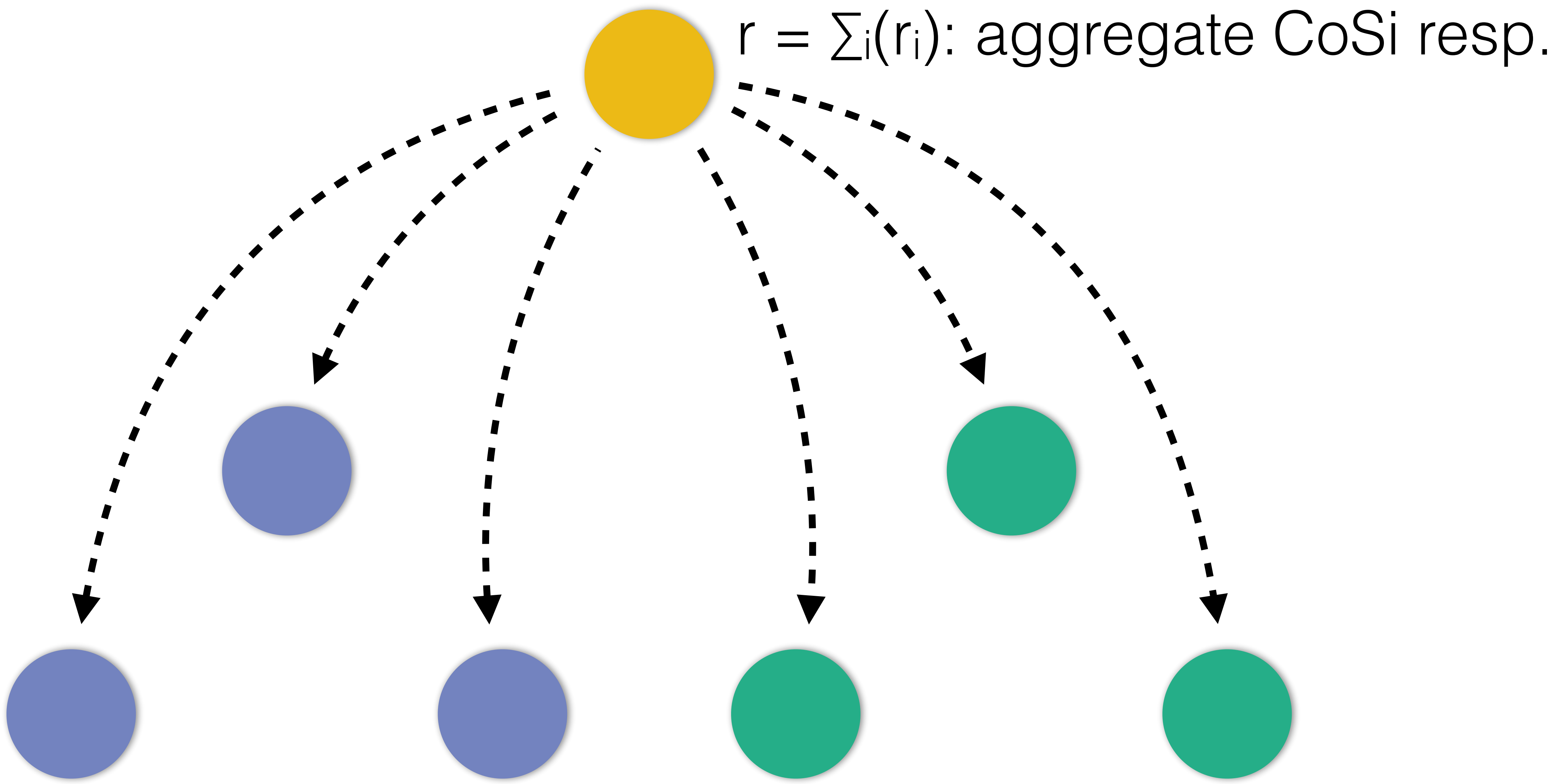
… …

$r_0 = v_0 - cx_0$ … … …

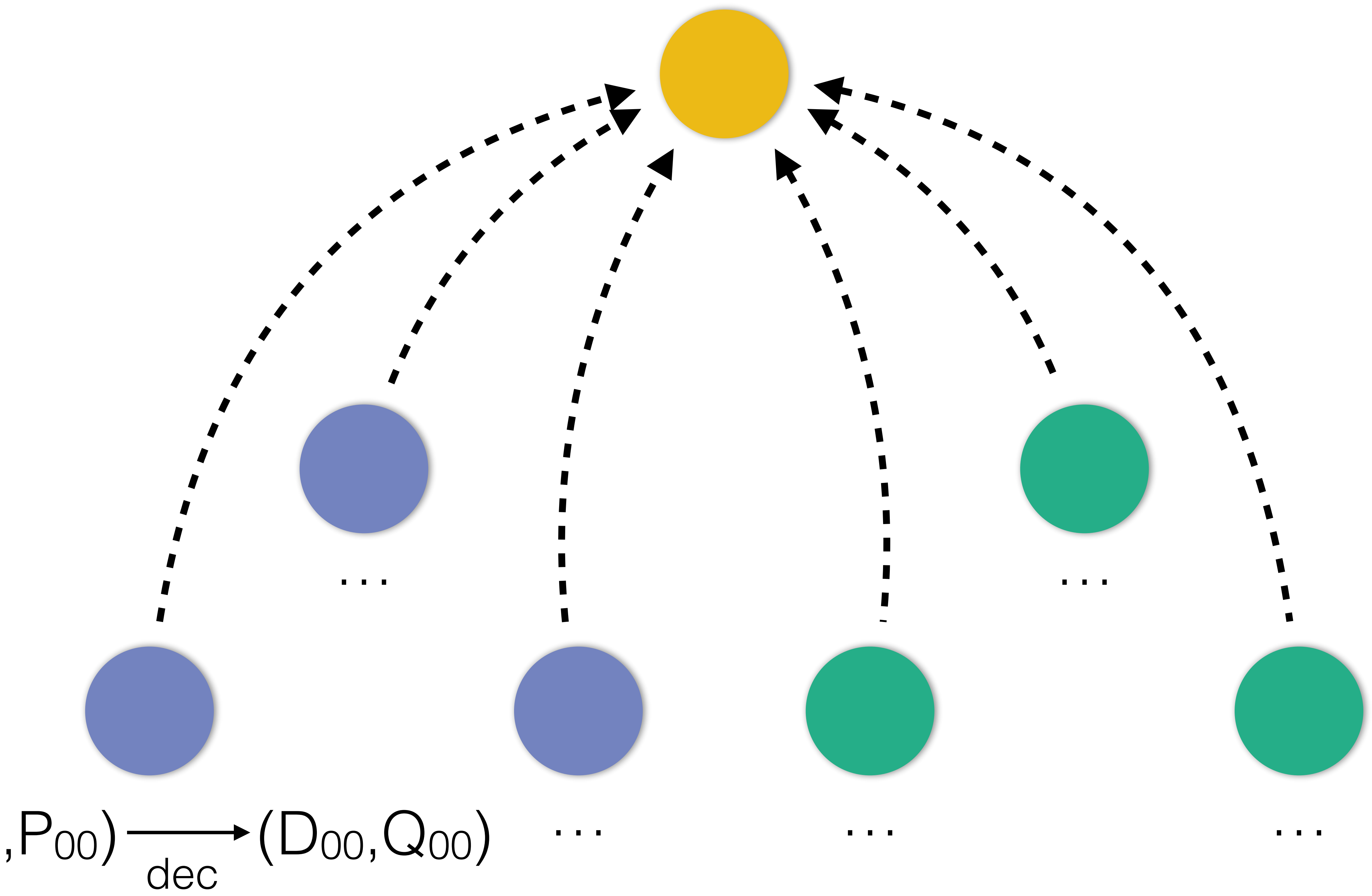
3. Secret Commitment (C)
Verify PVSS shares,
CoSi challenge: client commits to secrets

4. Secret Acknowledgement (S)
Verify commitment,
send (partial) CoSi responses

RandHound

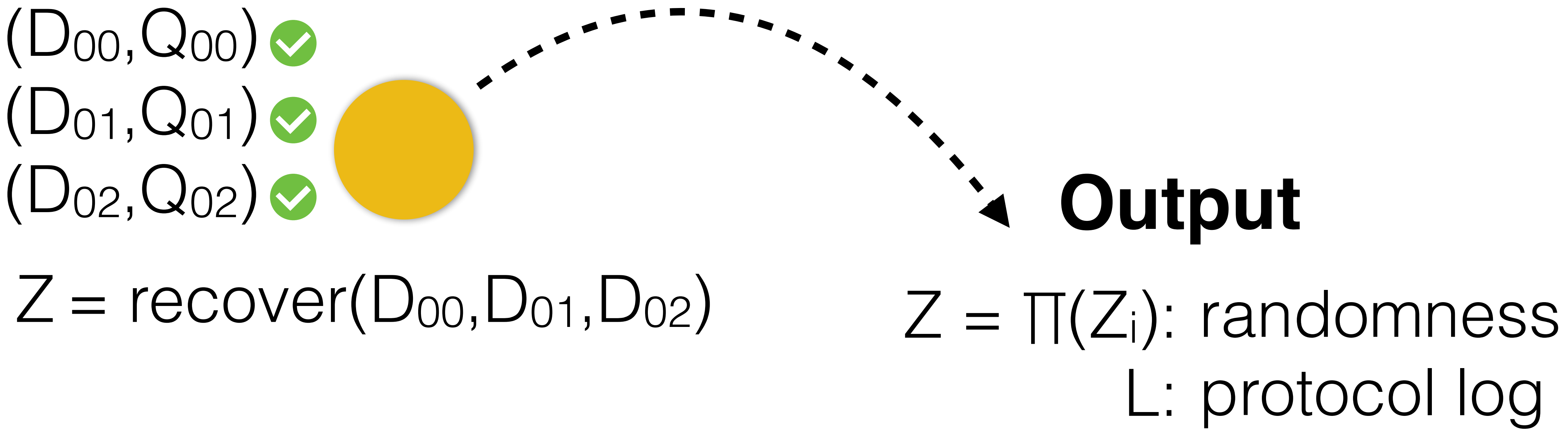


5. Decryption Request (C)
Request PVSS share decryption:
(aggregate) CoSi responses



6. Share Decryption (S)
Verify CoSi response,
If ok: decrypt valid PVSS shares

RandHound



(D_{00},Q_{00}) ✓
(D_{01},Q_{01}) ✓
(D_{02},Q_{02}) ✓

$Z = \text{recover}(D_{00},D_{01},D_{02})$

Output

$Z = \prod(Z_i)$: randomness
L: protocol log

7. Recover Randomness (C)
Verify decrypted PVSS shares,
compute collective randomness

Verify Randomness (anyone)

- Use a protocol log (transcript) L to verify randomness Z
- Replay and check all steps
- Accept if all correct

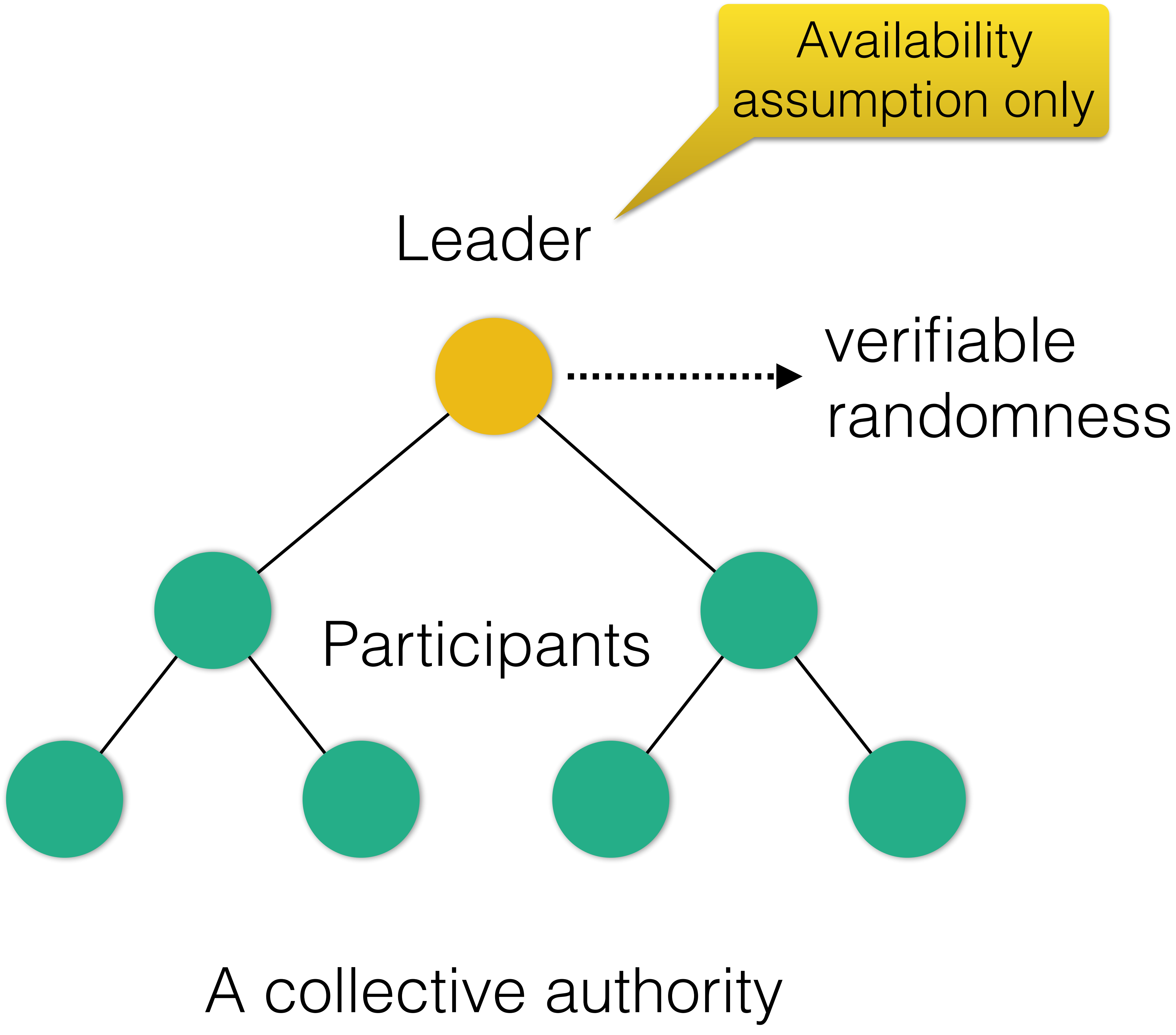
Public Randomness is (not so) Hard

	Availability	Unpredictability	Unbiasability	Verifiability	Scalability
Strawman I	⛔	⛔	⛔	⛔	⛔
Strawman II	⛔	✅	⛔	⛔	⛔
Strawman III	✅	✅	☑️	⛔	⛔
RandShare	✅	✅	✅	⛔	⛔
RandHound	✅	✅	✅	✅	☑️

Communication / computation complexity: $O(c^2n)$

RandHerd

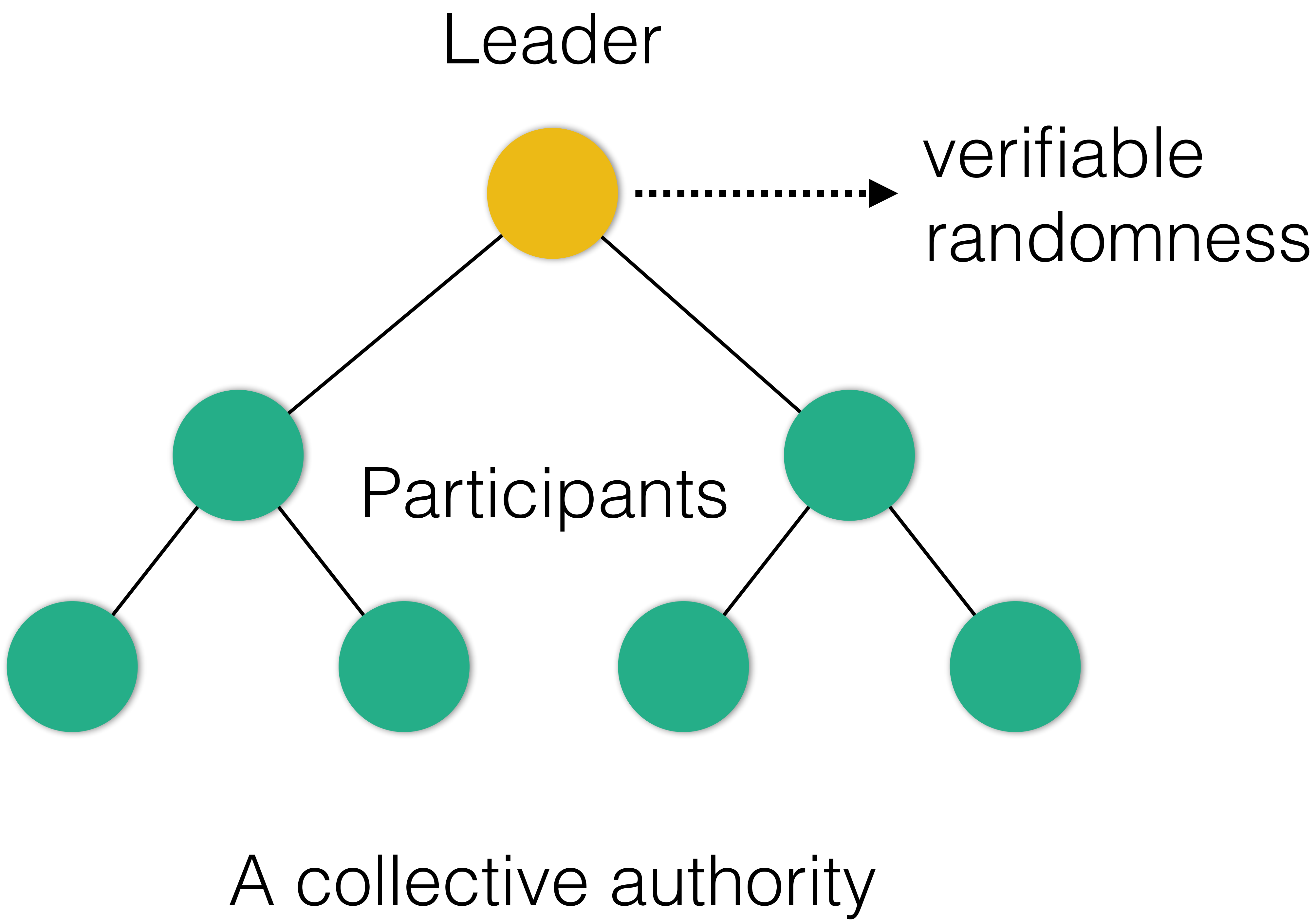
- Goals
  - Continuous, leader-coordinated randomness generation
  - Small randomness proof size (a single Schnorr signature)
  - Better performance than O(n)
- Decentralized randomness beacon
  - Built as a collective authority or *cothority*
  - Randomness on demand, at frequent intervals, or both



RandHerd

Achieving RandHerd's Goals

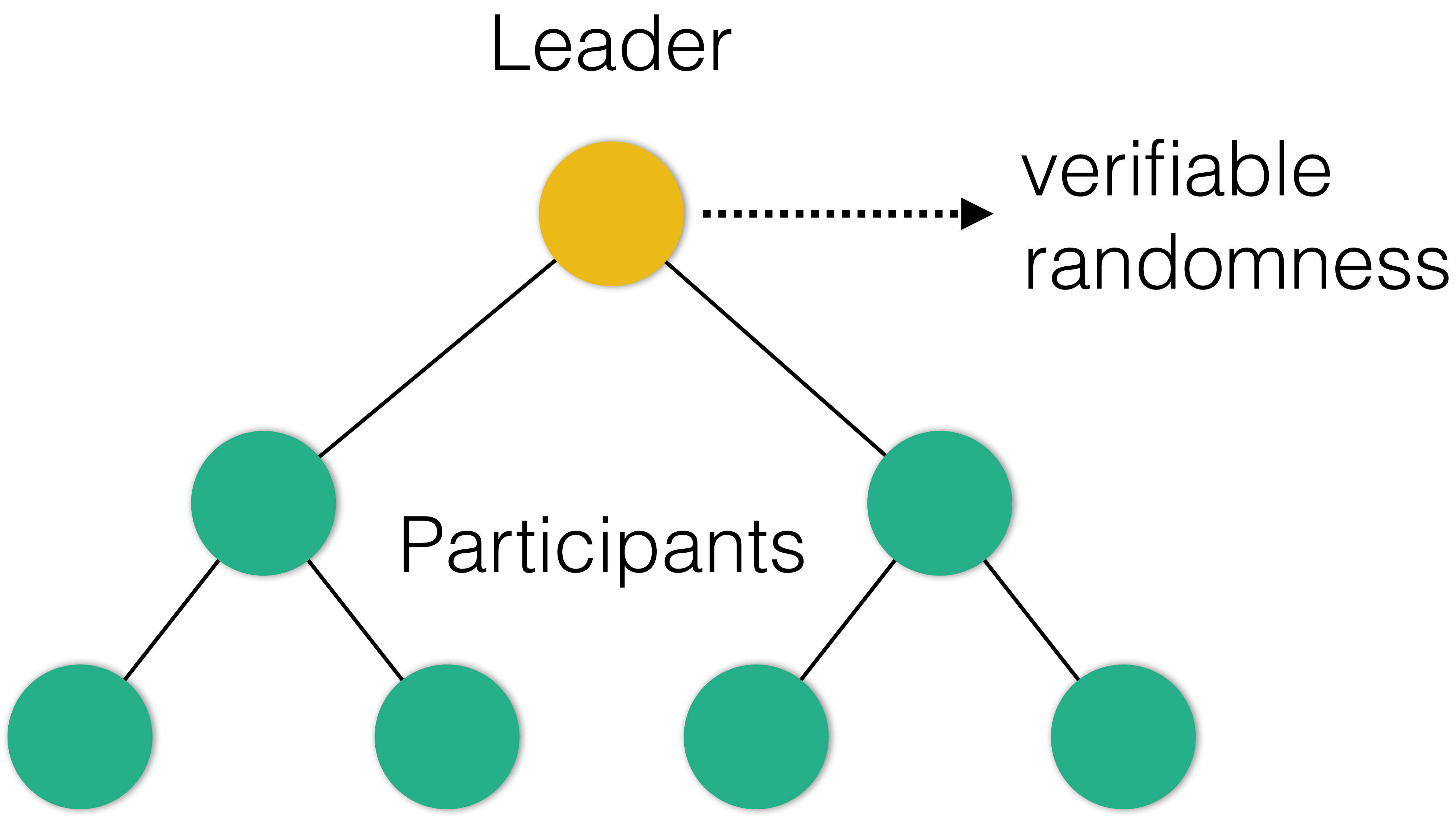
- Idea
  - Collective randomness = collective Schnorr signature
  - Benefits: Small proofs, O(log n) complexity
- Problem
  - Failing nodes influence output!
  - If some nodes unavailable, then the signature not a function of everyone's input



RandHerd

Achieving RandHerd's Goals

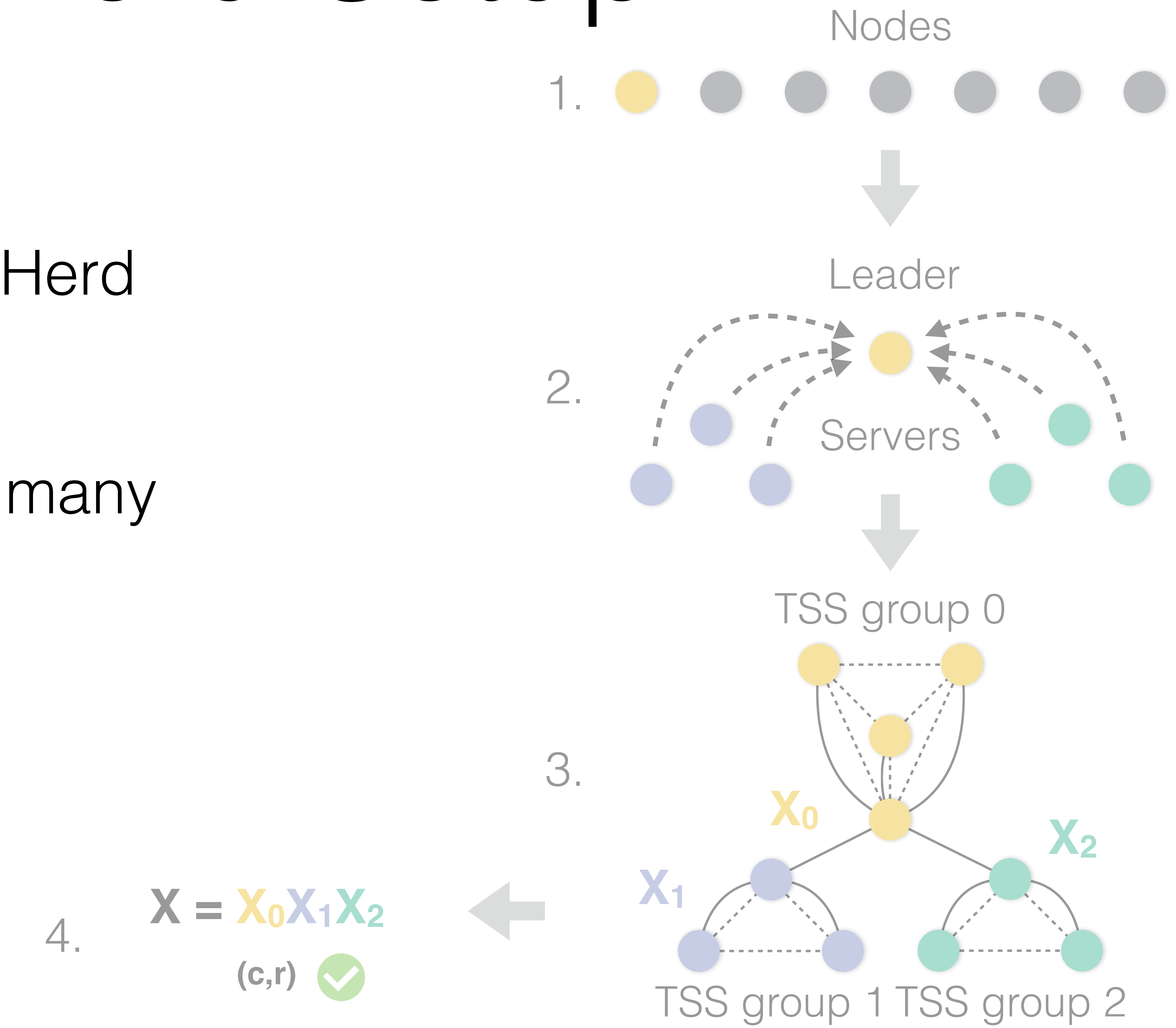
- Solution
  - Arrange nodes into *(t,n)*-threshold Schnorr signing (Stinson, 2001) groups (failure resistance)
  - Collective randomness = aggregate group signatures
  - Approach: Setup + round function



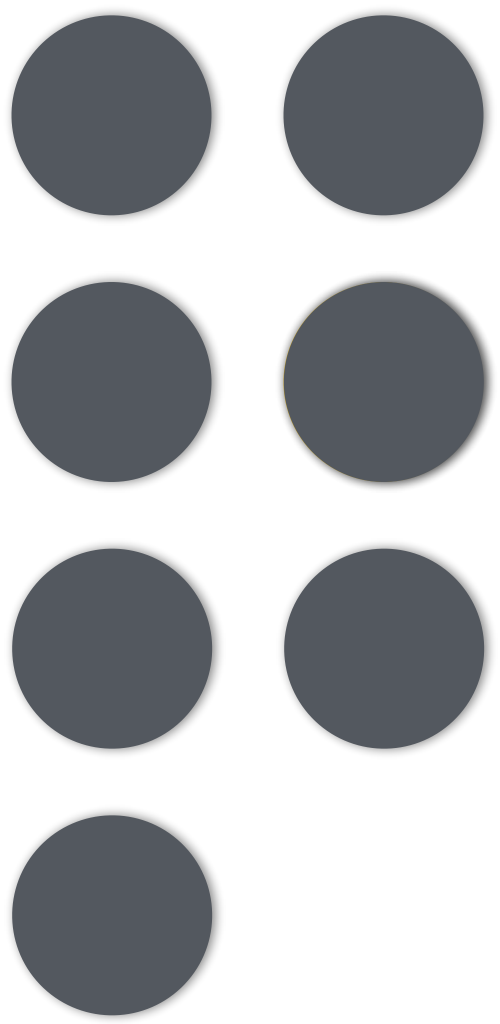
A collective authority

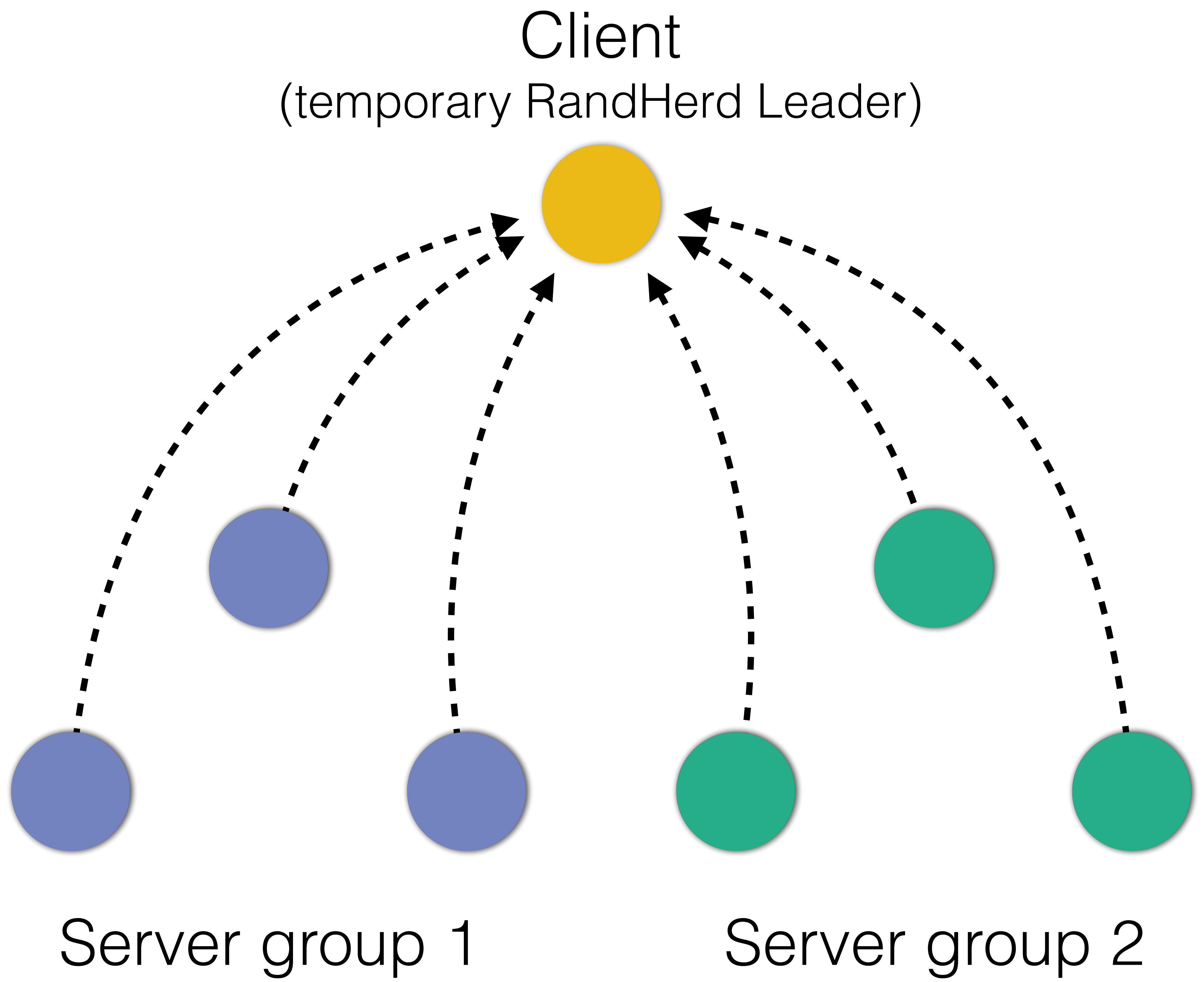
RandHerd Setup

- **Goal**: secure prep for RandHerd Round
- Executed once followed by many rounds of randomness
- Consists of 4 steps



RandHerd Setup

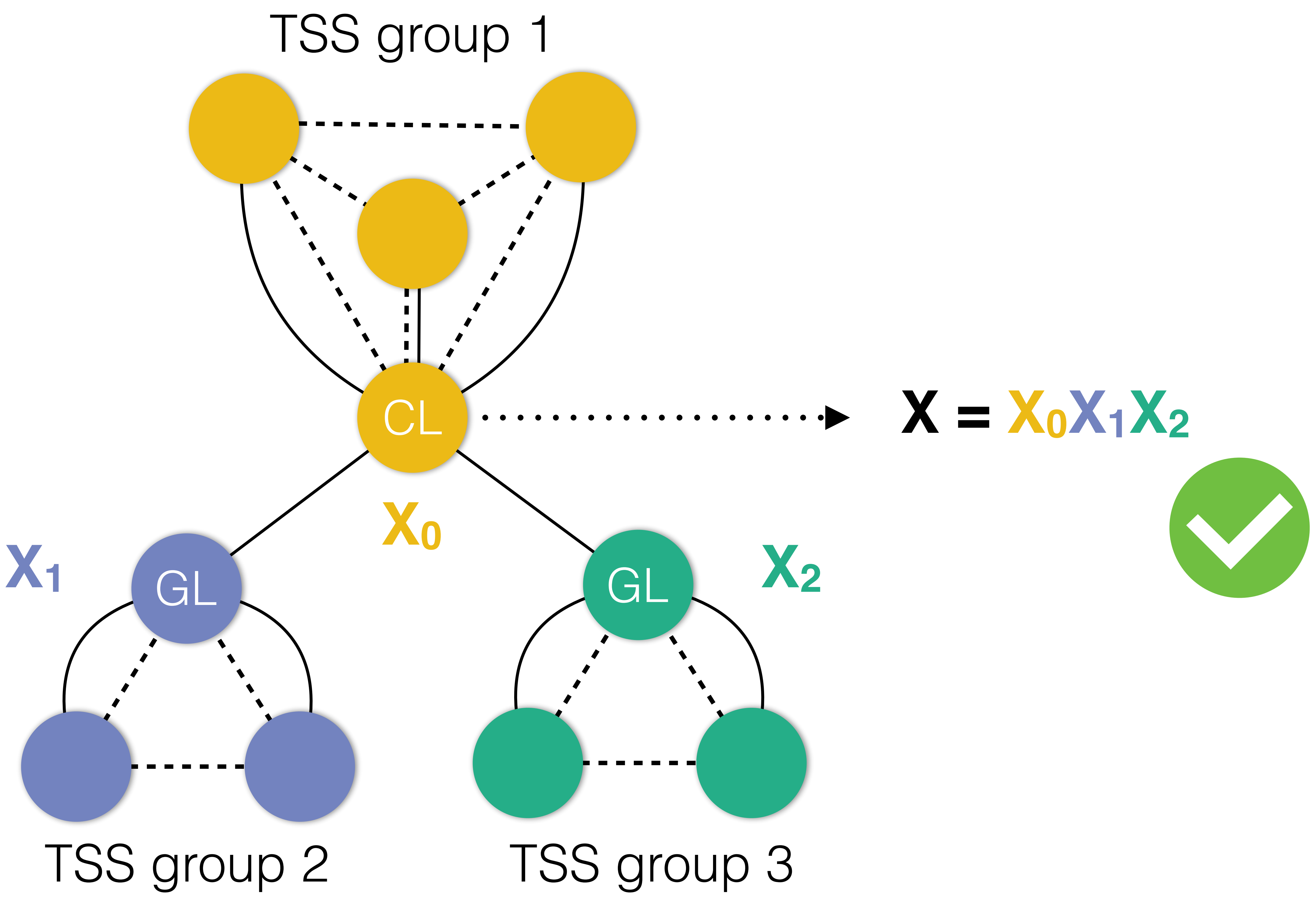




1. Leader Election
Elect a temporary leader via lowest ticket
t_i = VRF(*config*, *key*$_i$)

2. Sharding
Run RandHound to produce
(Z,L) as sharding seed

RandHerd Setup



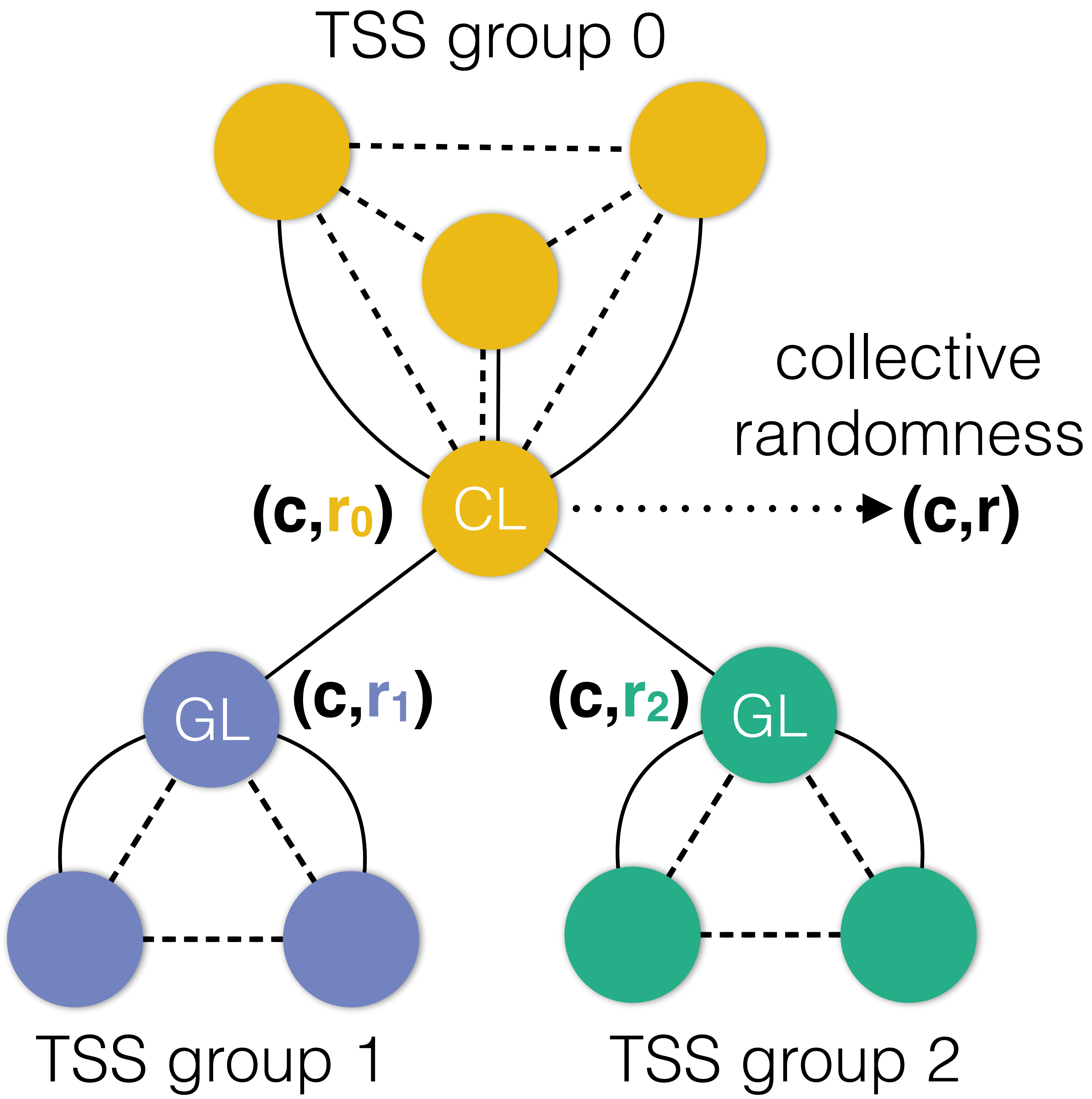
3. Group Setup
Create TSS groups using Z and generate group keys X_i

4. Collective RandHerd Key
Certify aggregate public key X using CoSi

RandHerd Round

Randomness Generation

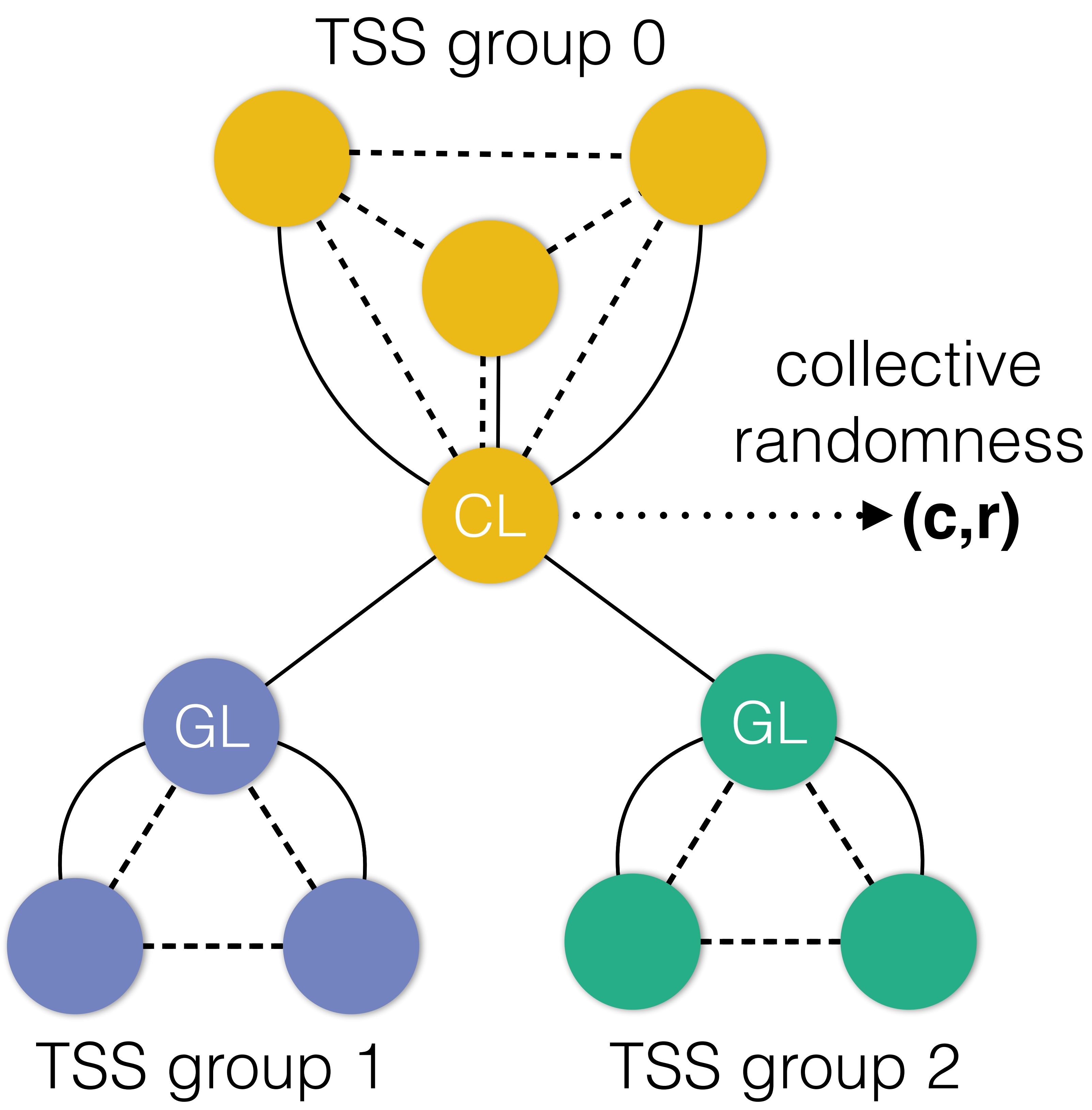
1. Cothority Leader (CL) broadcasts timestamp ***v***
2. TSS-CoSi
   a. Produce group Schnorr signatures **(c,r_0) (c,r_1) (c,r_2)** on ***v***
   b. At least 2f+1 nodes fix and certify challenge **c** using CoSi
   c. Aggregate into collective Schnorr signature **(c,$r = r_0+r_1+r_2$)**
   d. Publish **(c,r)** as collective randomness



RandHerd Round

Randomness Verification

1. RandHerd produces a simple Schnorr signature
2. Anyone can efficiently verify **(c,r)** on **v** using the collective public key $\mathbf{X} = \mathbf{X_0 X_1 X_2}$
3. Single signature verification!



Public Randomness is (not so) Hard

	Availability	Unpredictability	Unbiasability	Verifiability	Scalability
Strawman I	⛔	⛔	⛔	⛔	⛔
Strawman II	⛔	✅	⛔	⛔	⛔
Strawman III	✅	✅	☑	⛔	⛔
RandShare	✅	✅	✅	⛔	⛔
RandHound	✅	✅	✅	✅	☑
RandHerd	✅	✅	✅	✅	✅

Communication / computation complexity: $O(c^2\log(n))$

Talk Outline

- Motivation
- Two Randomness Protocols
    - RandHound
    - RandHerd
- **Implementation, Experimental Results and Current Deployment**
- Conclusions

Implementation & Experiments

Implementation

- Go versions of DLEQ-proofs, PVSS, TSS, CoSi-TSS, RandHound, RandHerd

- Based on DEDIS code
  - Crypto library
  - Network library
  - Cothority framework

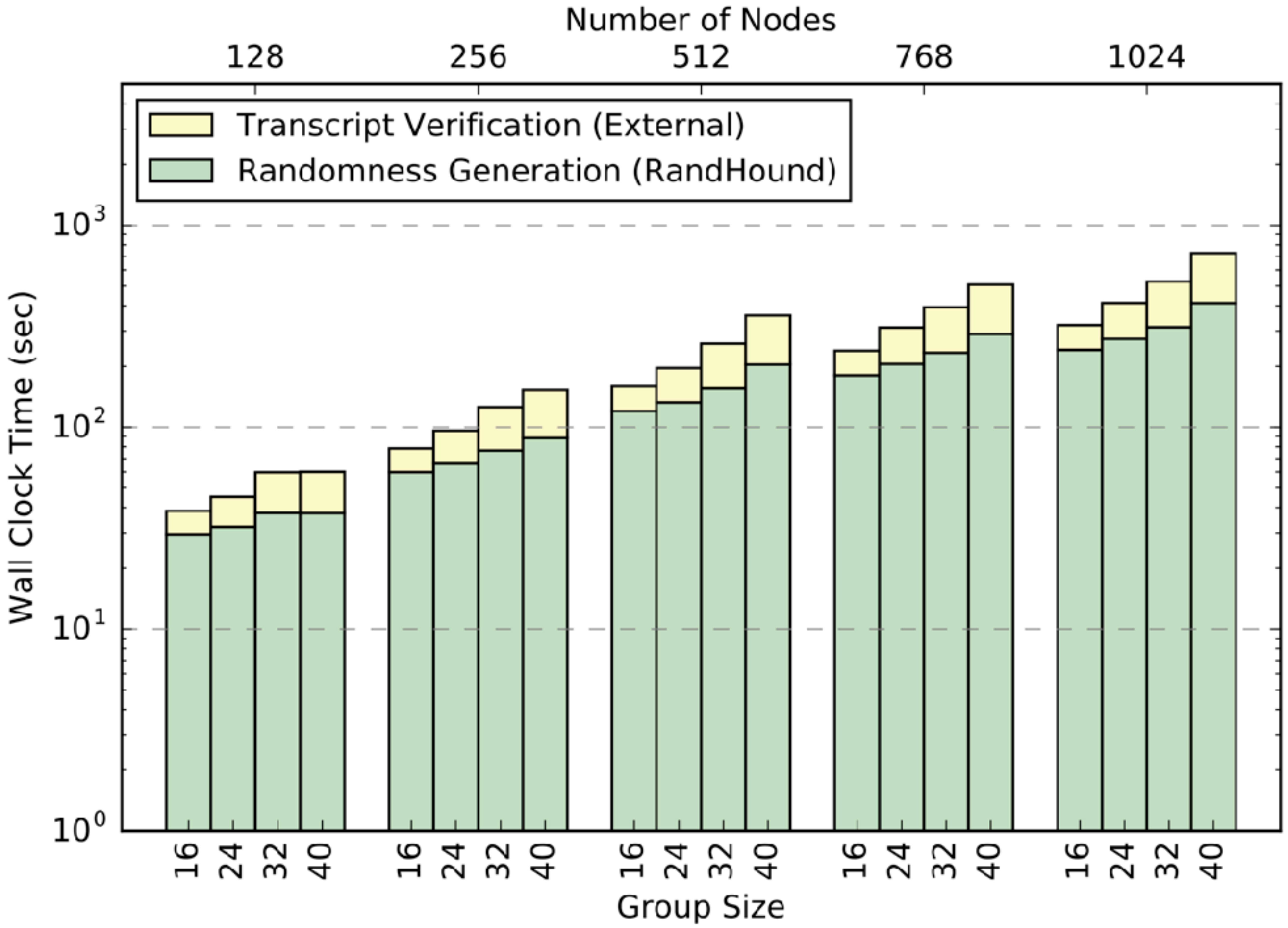
- https://github.com/dedis

DeterLab Setup

- 32 physical machines
  - Intel Xeon E5-2650 v4 (24 cores @ 2.2 GHz)
  - 64 GB RAM
  - 10 Gbps network link

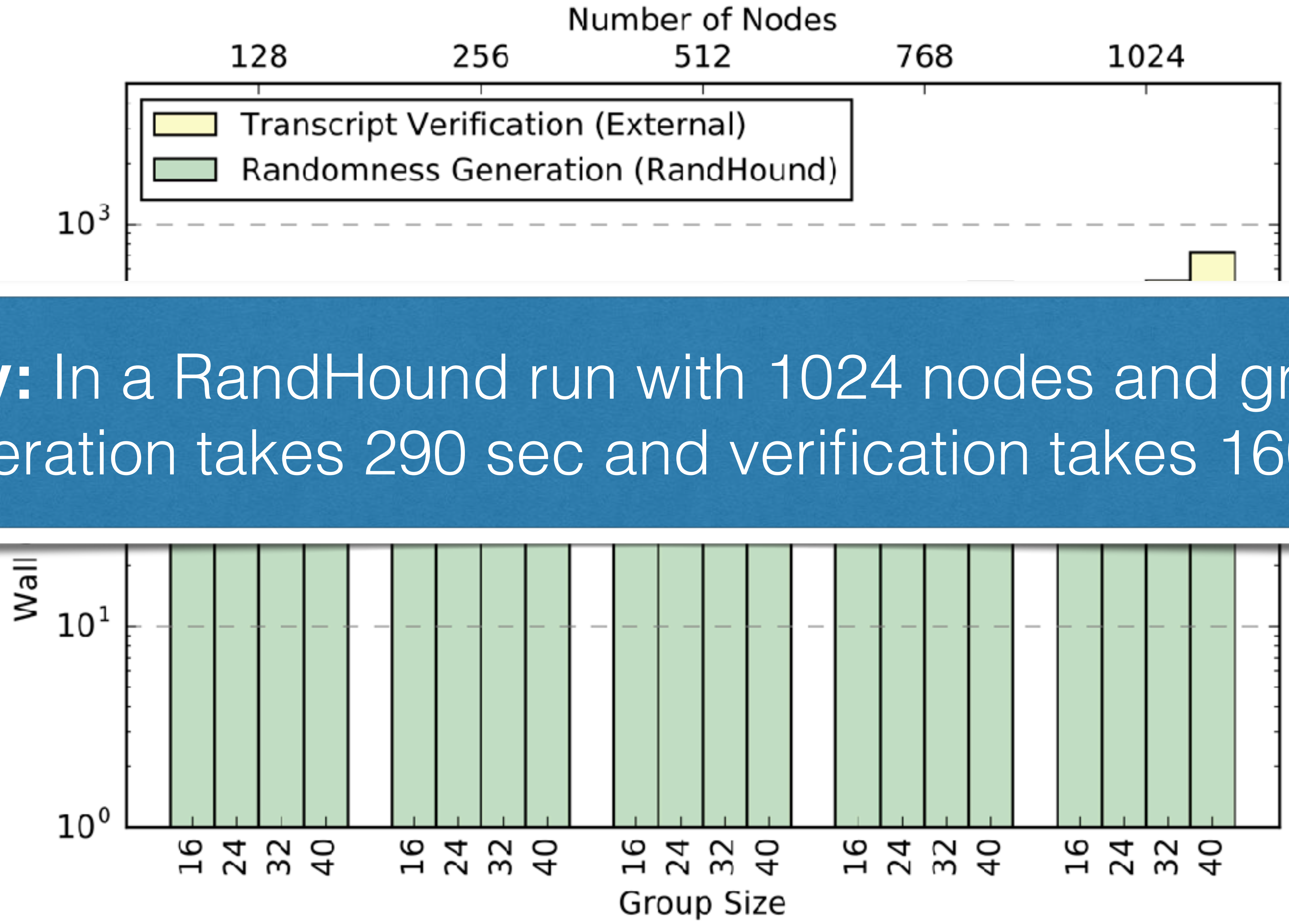
- Network restrictions
  - 100 Mbps bandwidth
  - 200 ms round-trip latency

Experimental Results – RandHound



Randomness generation and verification time

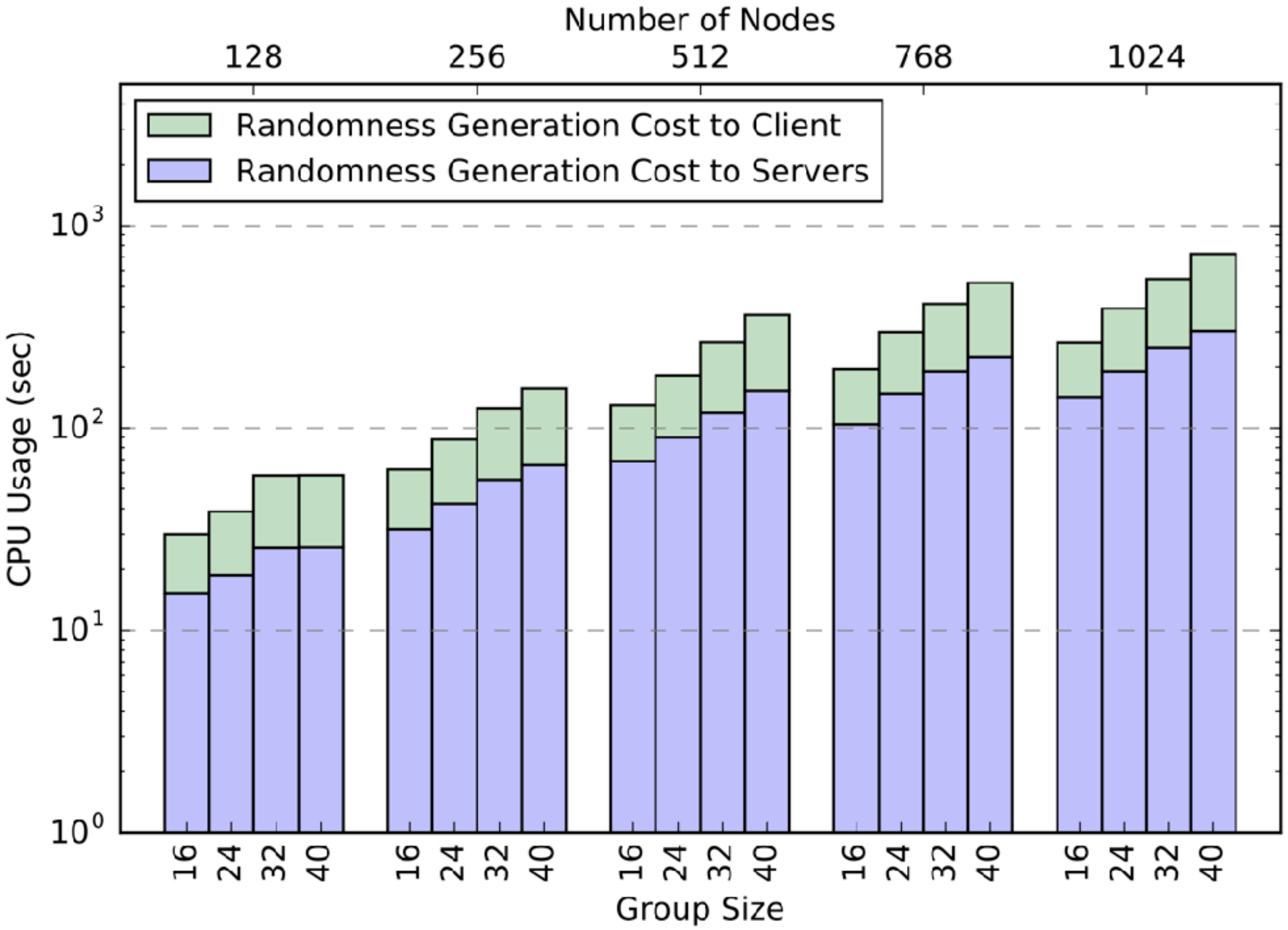
Experimental Results – RandHound



Take-away: In a RandHound run with 1024 nodes and group size 32, generation takes 290 sec and verification takes 160 sec.

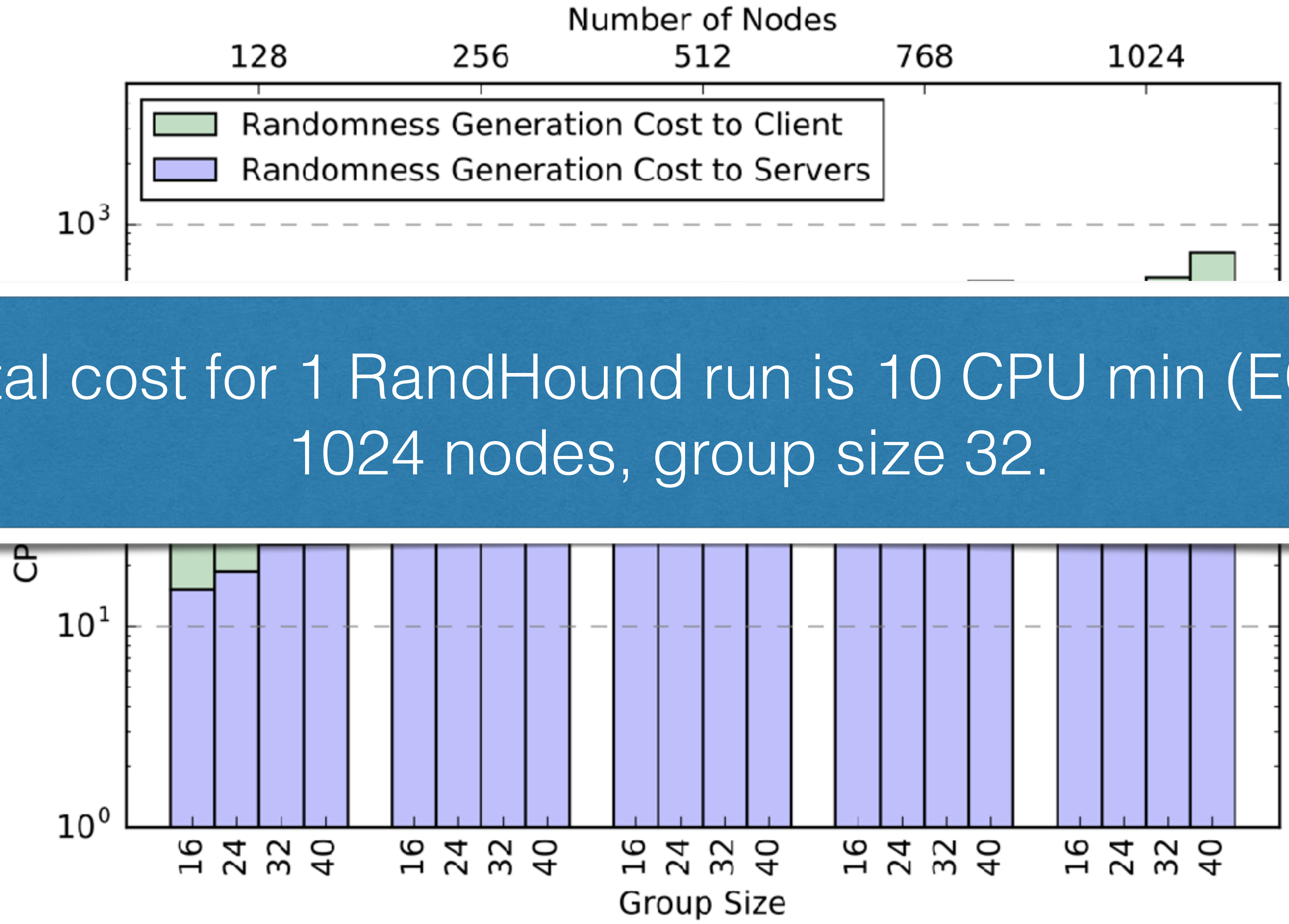
Randomness generation and verification time

Experimental Results – RandHound



CPU cost for the client and the servers

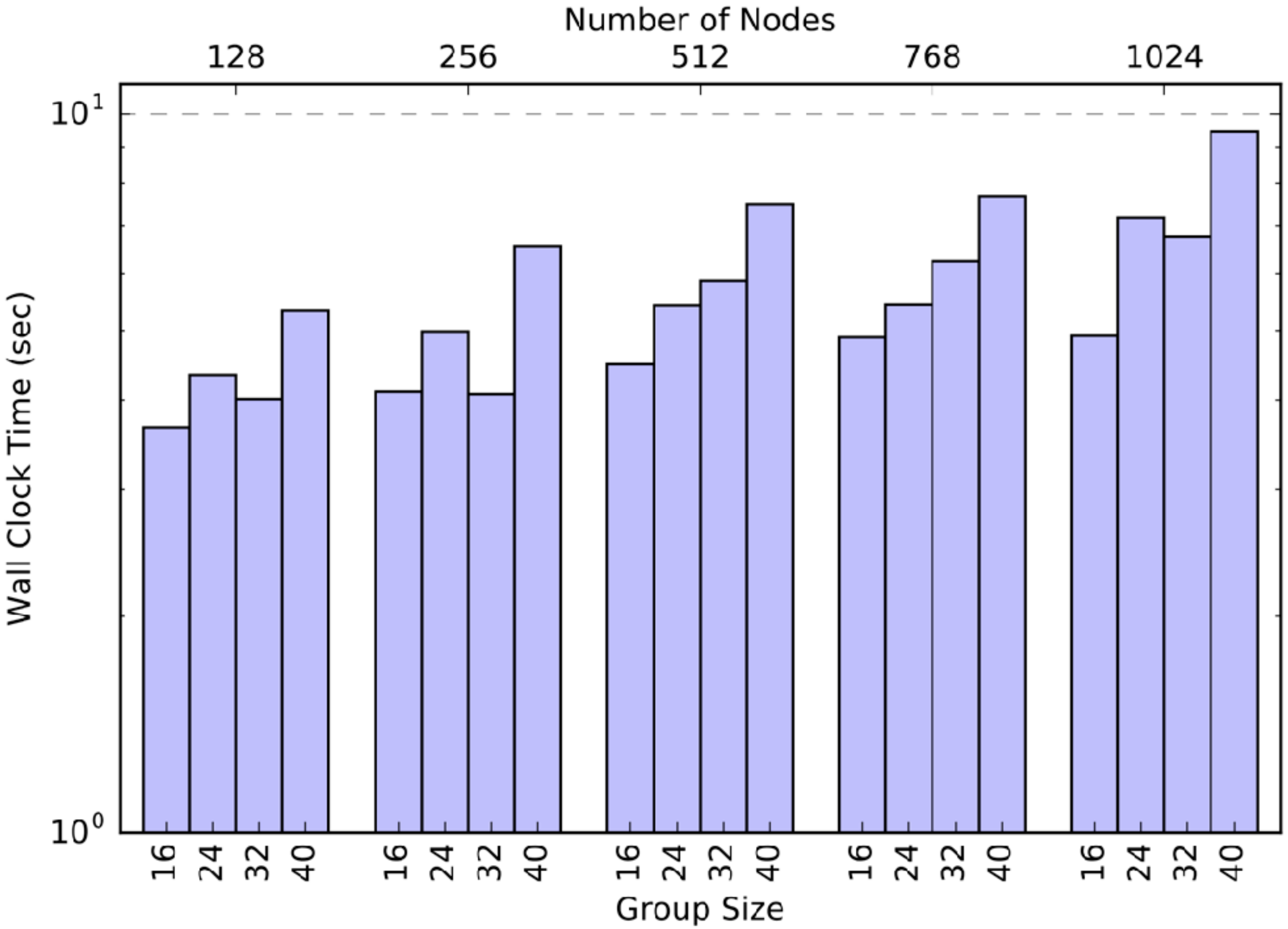
Experimental Results – RandHound



Take-away: Total cost for 1 RandHound run is 10 CPU min (EC2: < $0.02) with 1024 nodes, group size 32.

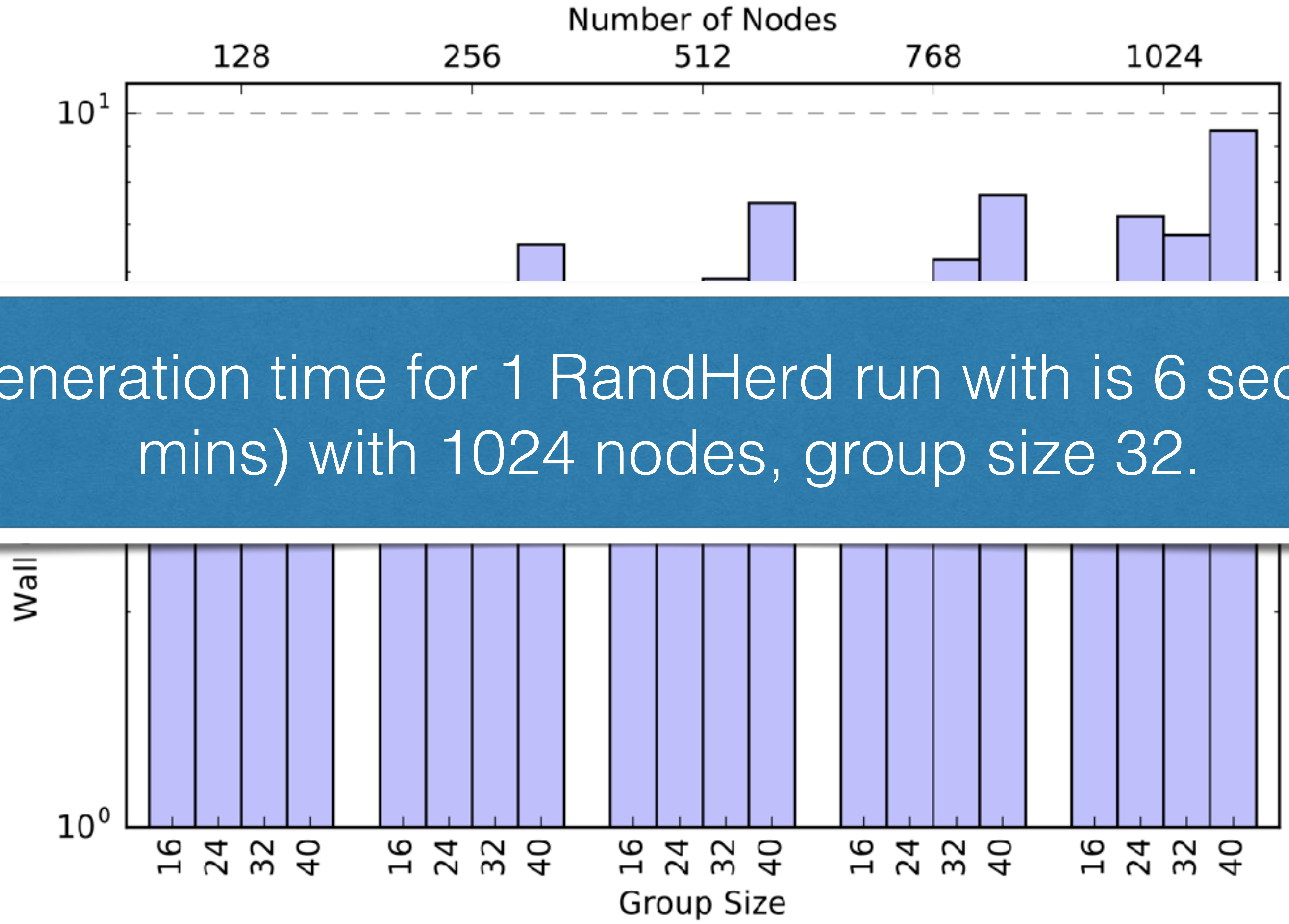
CPU cost for the client and the servers

Experimental Results – RandHerd



Randomness generation time

Experimental Results – RandHerd



Take-away: Generation time for 1 RandHerd run with is 6 sec, after setup (10 mins) with 1024 nodes, group size 32.

Randomness generation time

drand

Proof-of-concept Randomness-as-a-Service

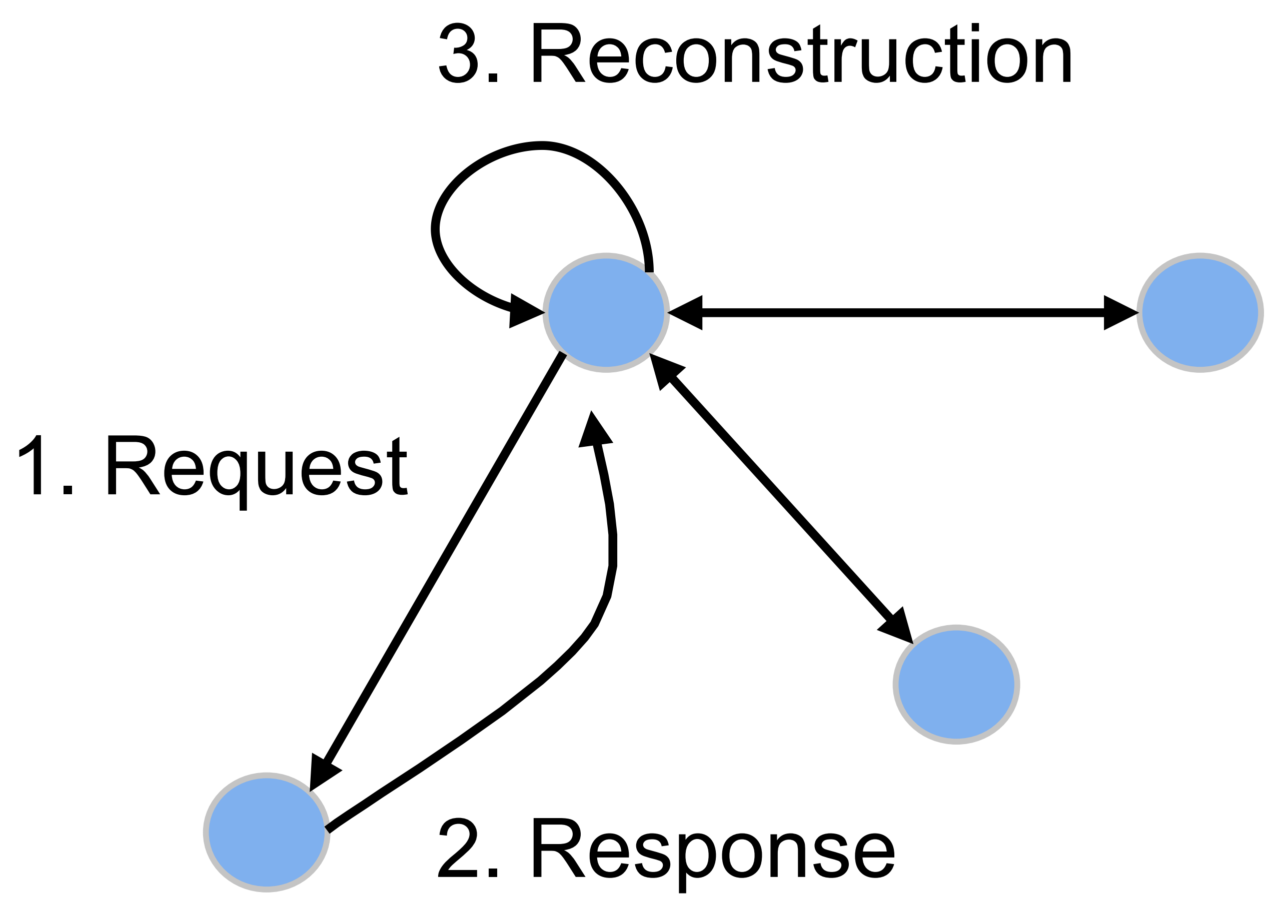
https://github.com/dedis/drand

Nicolas Gailly
nicolas.gailly@epfl.ch

drand: the protocol

- Implements the **logic** of RandHerd (leaderless, pairing-based crypto)

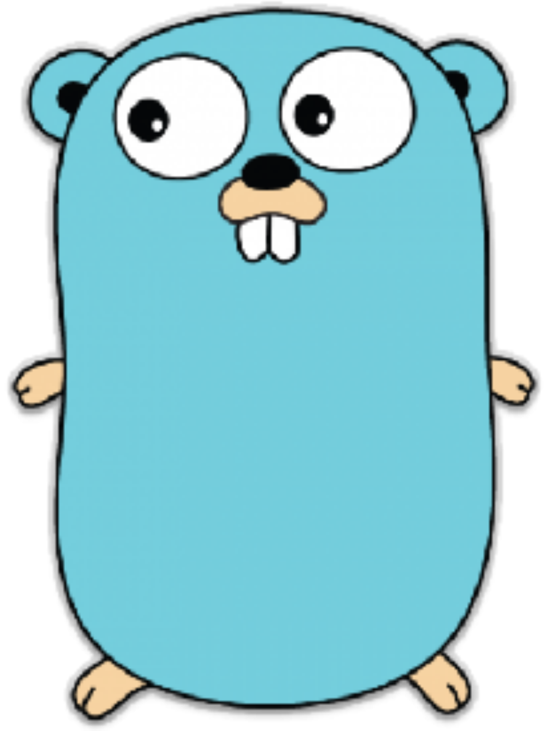
- Setup
  - Threshold Distributed Key Generation (DKG) (Gennaro, 2007)
  - Collective public key and each node has a share of the private key
  - Can refresh shares without changing the public key (Wong, 2002)



- Randomness Generation
  - Threshold Boneh-Lynn-Shacham (BLS) signature
  - Each node **requests** a partial signature, waits for at least *t* **responses** and **reconstructs**
  - First sign fixed seed and then the randomness from previous round

drand: the software

- Implemented in Go, open source (on GitHub)
- Meant to be very simple
  - 1 command for setup as well as generation
  - JSON API to fetch randomness (browser!)
  - Docker container provided
- Deployment
  - EPFL, NIST, Cloudflare, Kudelski Security and hopefully others to run drand nodes



Conclusion

- Generation of public randomness: **trust** and **scale** issues

- Our solution: two protocols in the *(t,n)*-threshold security model

	Availability	Unpredictability	Unbiasability	Verifiability	Scalability	Complexity
RandHound	✓	✓	✓	✓	✓	O(n)
RandHerd	✓	✓	✓	✓	✓	O(log(n))

- Code: https://github.com/dedis/cothority

Talk Outline

Scalable Bias-Resistant Distributed Randomness

2017 IEEE Symposium on Security and Privacy

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

2018 IEEE Symposium on Security and Privacy

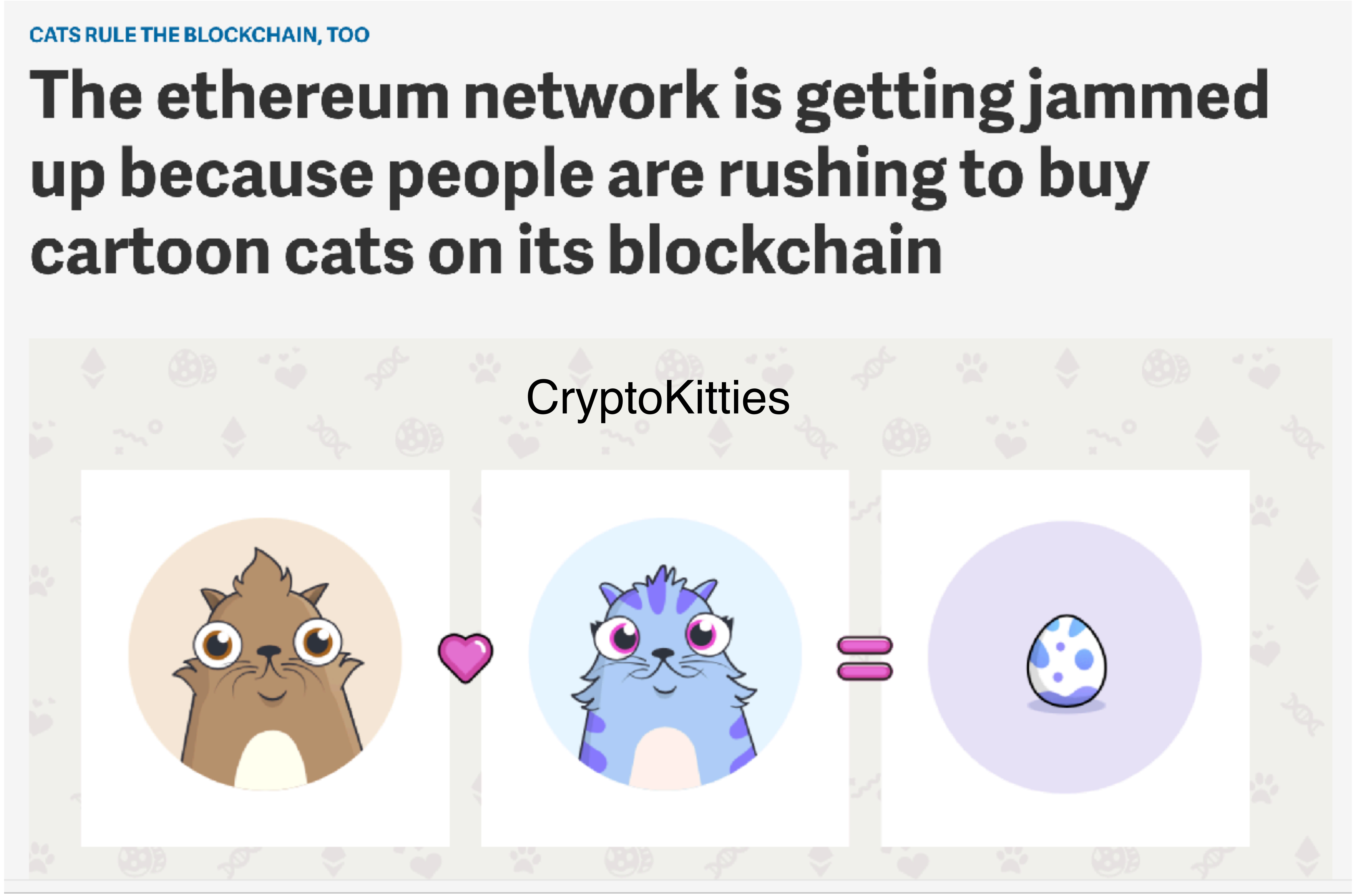
Talk Outline

- Motivation
- OmniLedger
- Evaluation
- Conclusion

Talk Outline

- **Motivation**
- OmniLedger
- Evaluation
- Conclusion

Scaling Blockchains is More Important Than Ever …



CATS RULE THE BLOCKCHAIN, TOO

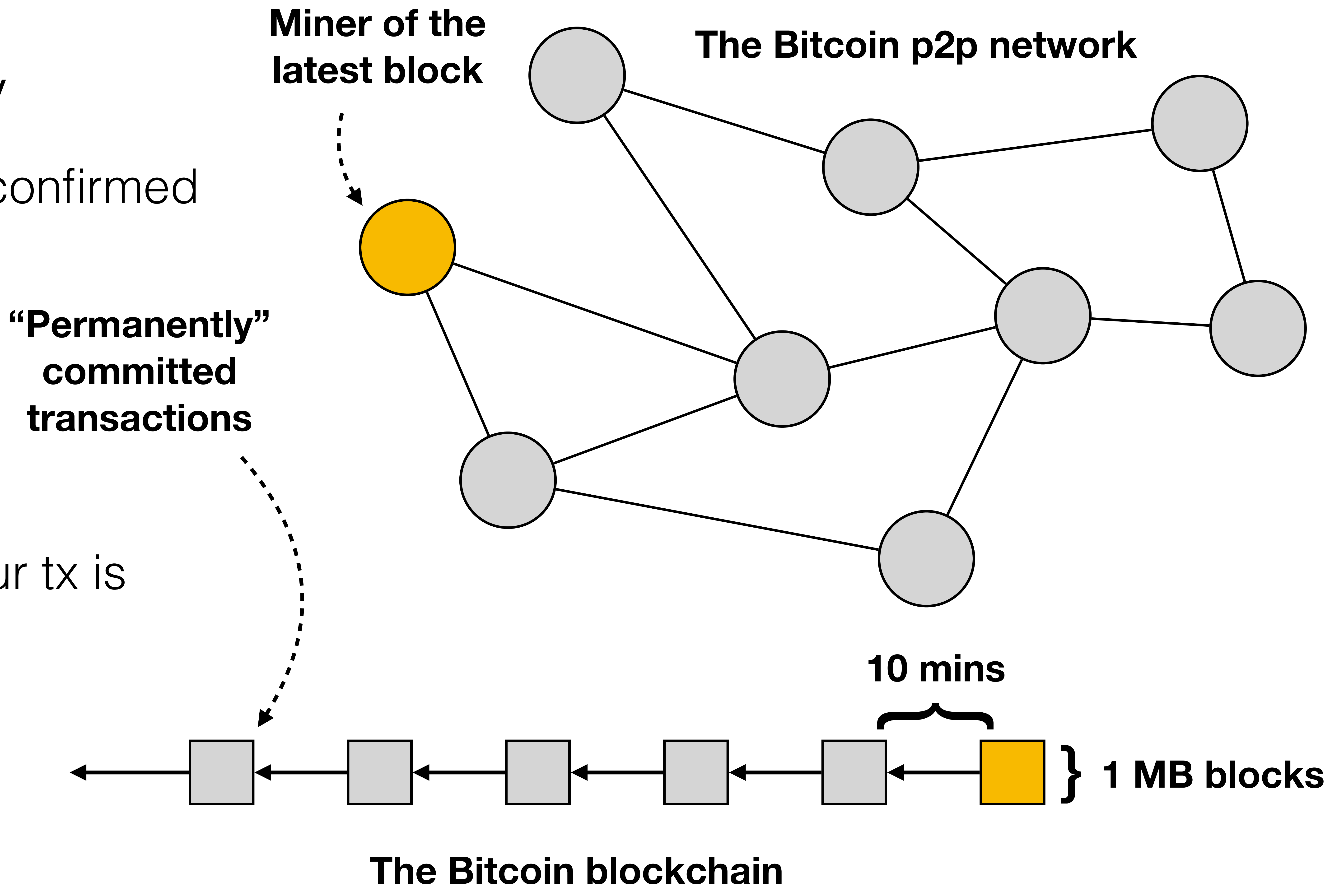
The ethereum network is getting jammed up because people are rushing to buy cartoon cats on its blockchain

CryptoKitties

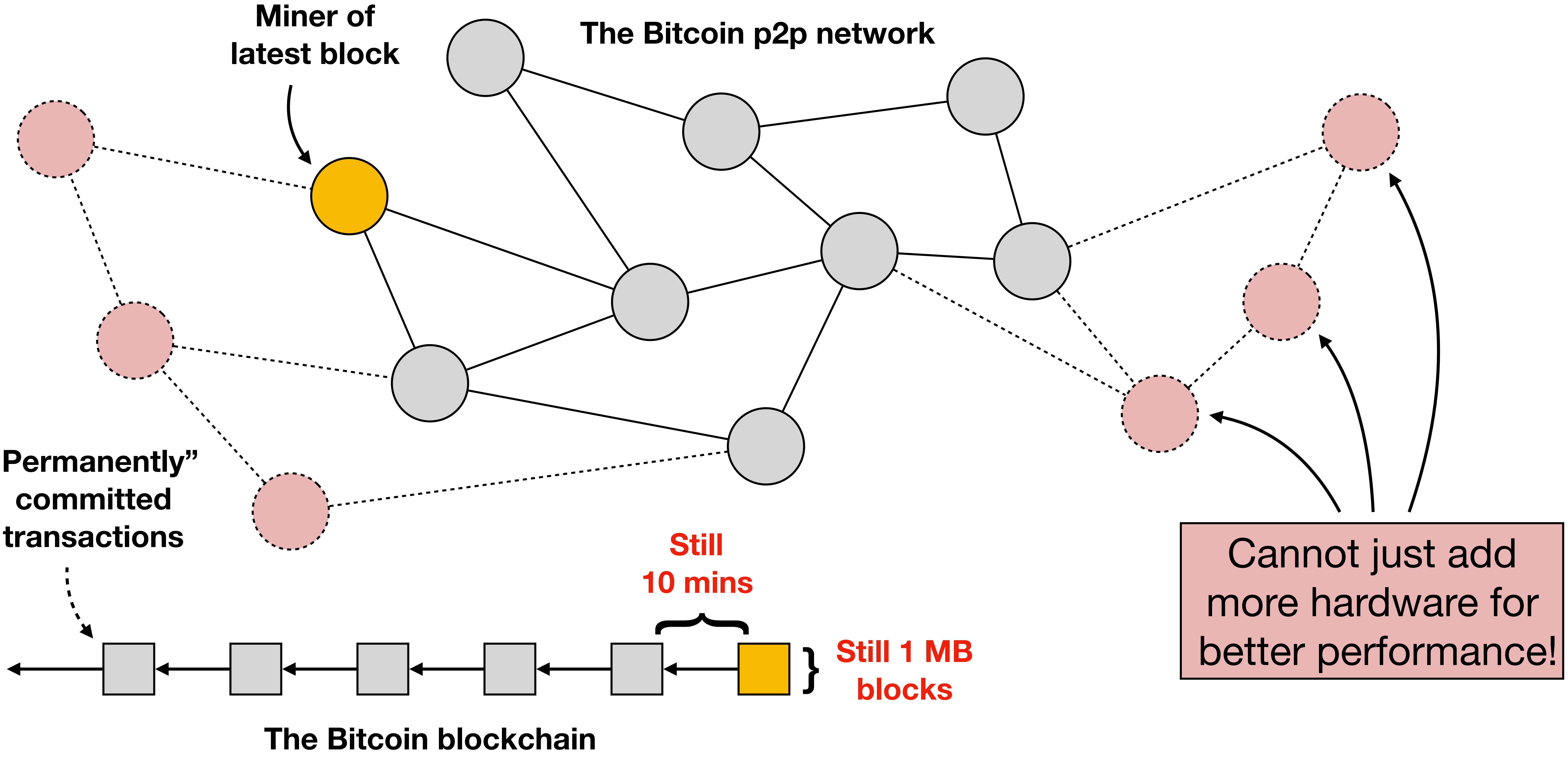
The Core of Bitcoin: Nakamoto Consensus

Drawbacks

- Transaction confirmation delay
  - Bitcoin: Any tx takes >10 mins until confirmed
- Low throughput
  - Bitcoin: ~4 tx/sec
- Weak consistency
  - Bitcoin: You are not really certain your tx is committed until you wait >1 hour
- Proof-of-work mining
  - Wastes huge amount of energy



… But Scaling Blockchains is Not Easy



Miner of latest block

The Bitcoin p2p network

“Permanently” committed transactions

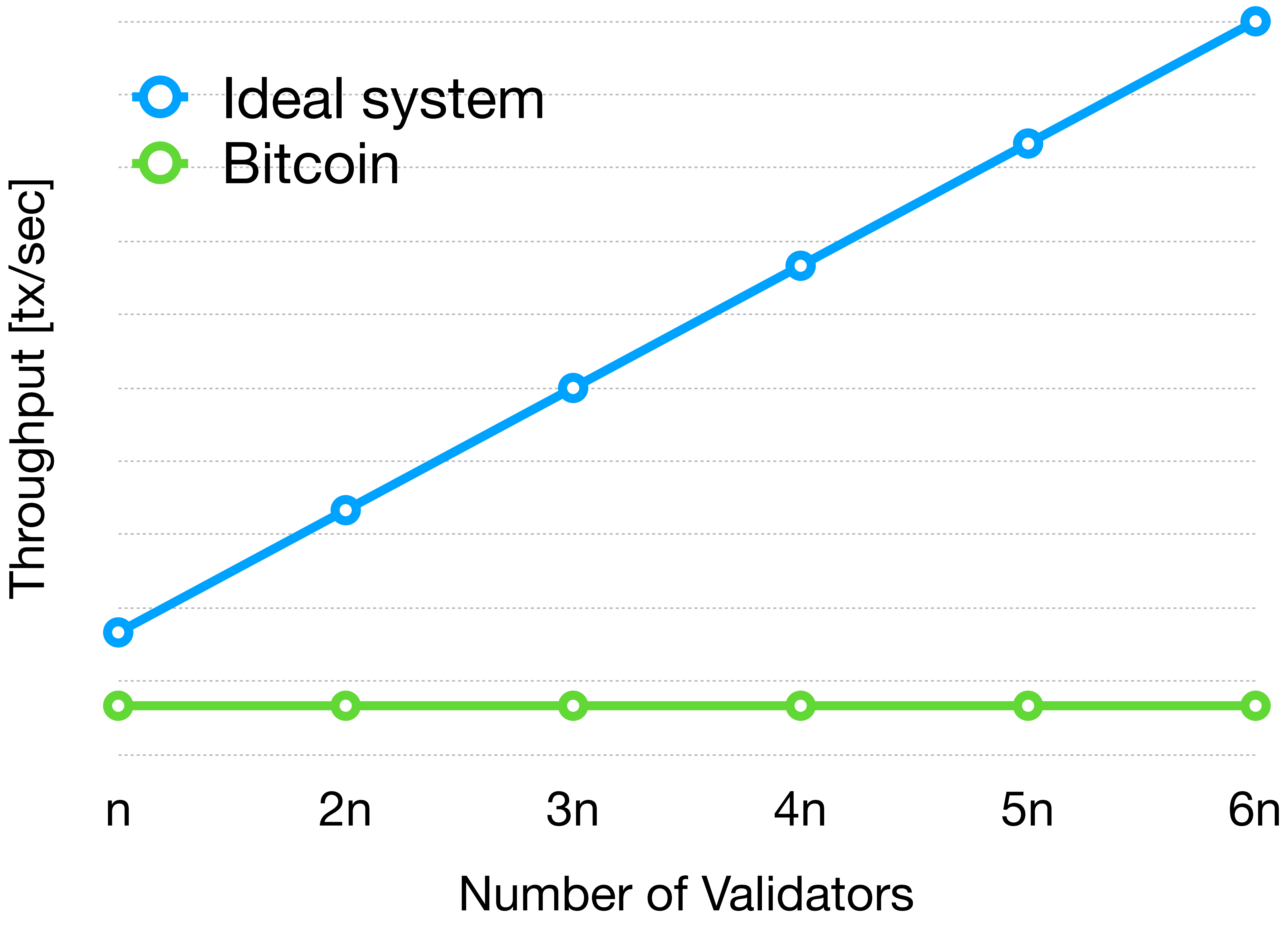
Still 10 mins

Still 1 MB blocks

The Bitcoin blockchain

Cannot just add more hardware for better performance!

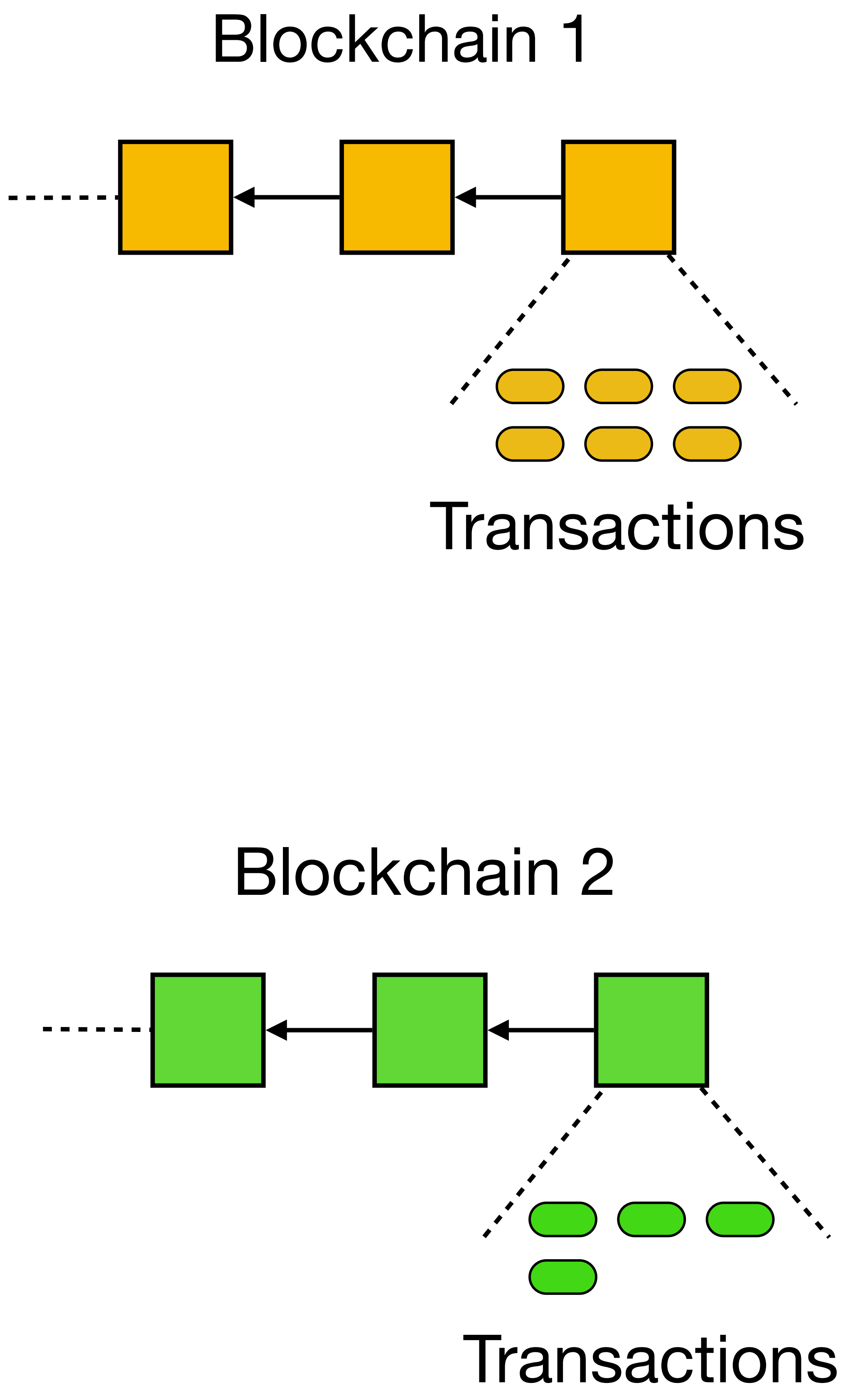
What we Want: Scale-Out Performance



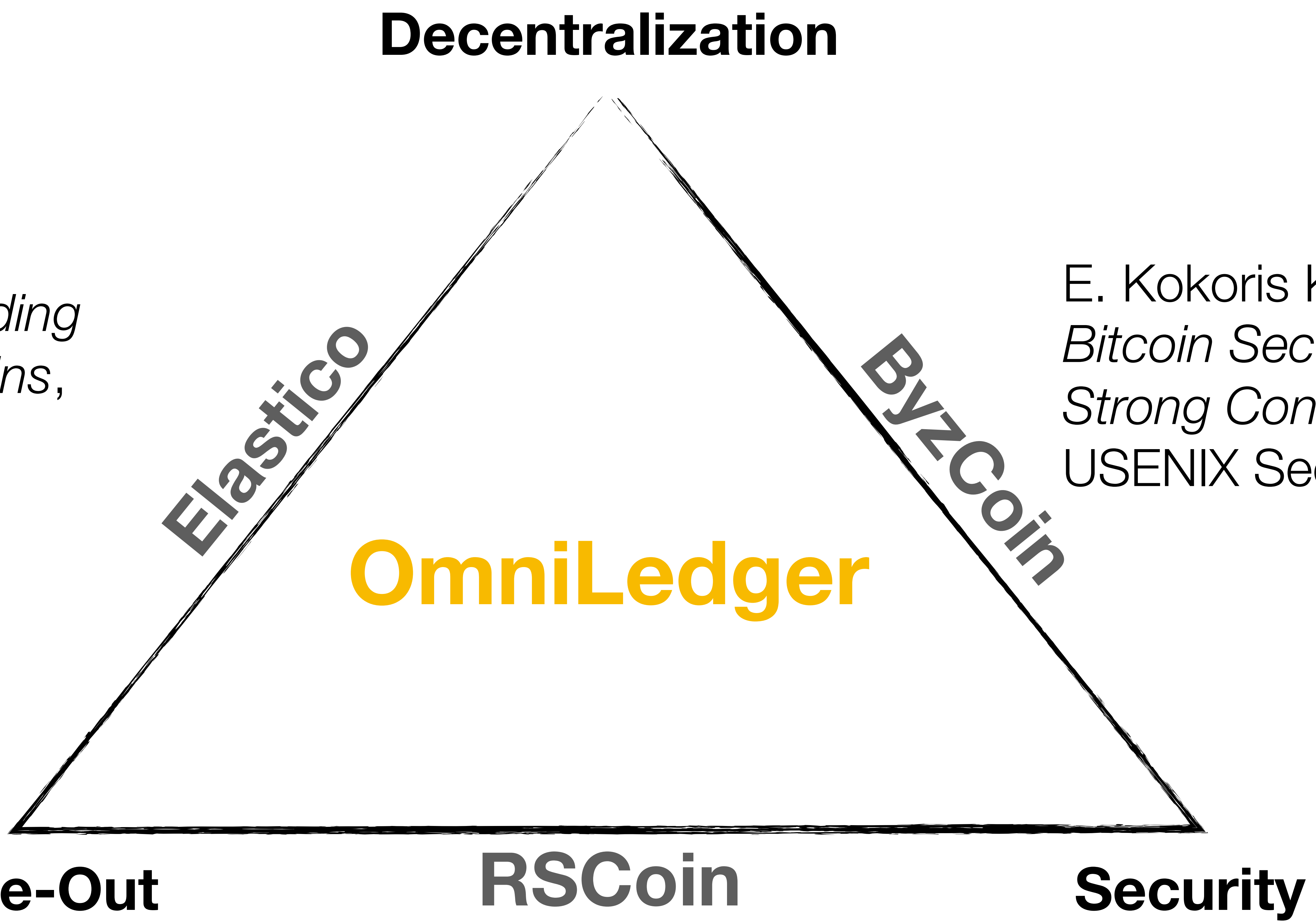
Scale-out: Throughput increases *linearly* with the available resources.

Towards Scale-Out Performance via Sharding

- **Concept:**
  - Validators are grouped into distinct subsets
  - Each subset processes different transactions
  - Achieves parallelization and therefore scale-out
- **But:**
  - How to assign validators to shards?
  - How to send transactions across shards?



Distributed Ledger Landscape



L. Luu et al., *A Secure Sharding Protocol for Open Blockchains*, CCS 2016

E. Kokoris Kogias et al., *Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing*, USENIX Security 2016

G. Danezis and S. Meiklejohn, *Centrally Banked Cryptocurrencies*, NDSS 2016

Talk Outline

- Motivation
- **OmniLedger**
- Evaluation
- Conclusion

OmniLedger – Design Goals

Security Goals

1. Full Decentralization
No trusted third parties or single points of failure

2. Shard Robustness
Shards process txs correctly and continuously

3. Secure Transactions
Txs commit atomically or abort eventually

Performance Goals

4. Scale-out
Throughput increases linearly in the number of active validators

5. Low Storage
Validators do not need to store the entire shard tx history

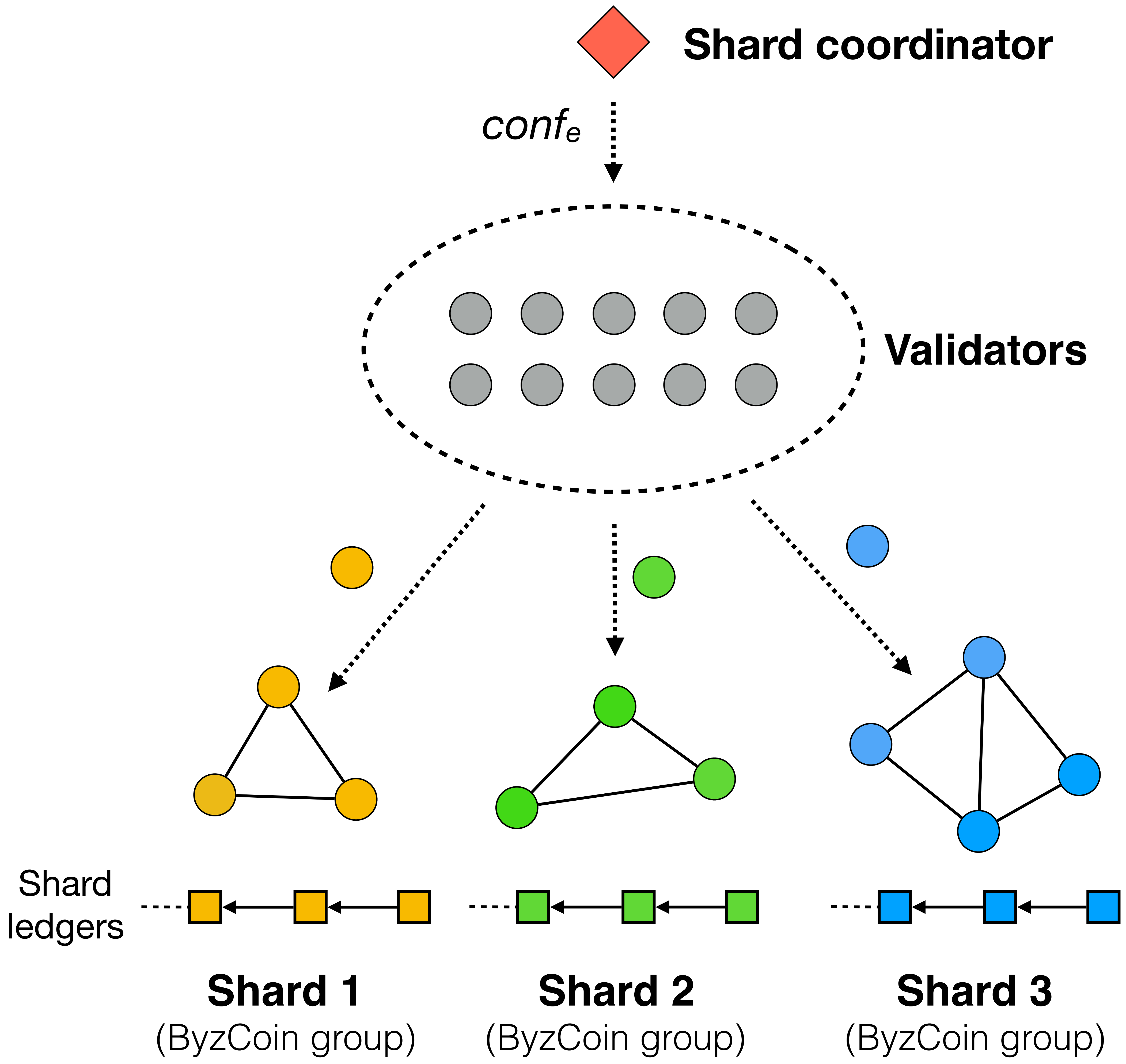
6. Low Latency
Tx are confirmed quickly

Assumptions: <= 25% mildly adaptive Byzantine adversary, (partially) synchronous network, UTXO model

Strawman: SimpleLedger

Overview

- Evolves in epochs *e*
- Trusted source releases shard configuration $conf_e$
- Validators:
  - Bootstrap from the shard ledger according to $conf_e$
  - Process transactions in parallel using per-shard consensus (ByzCoin)



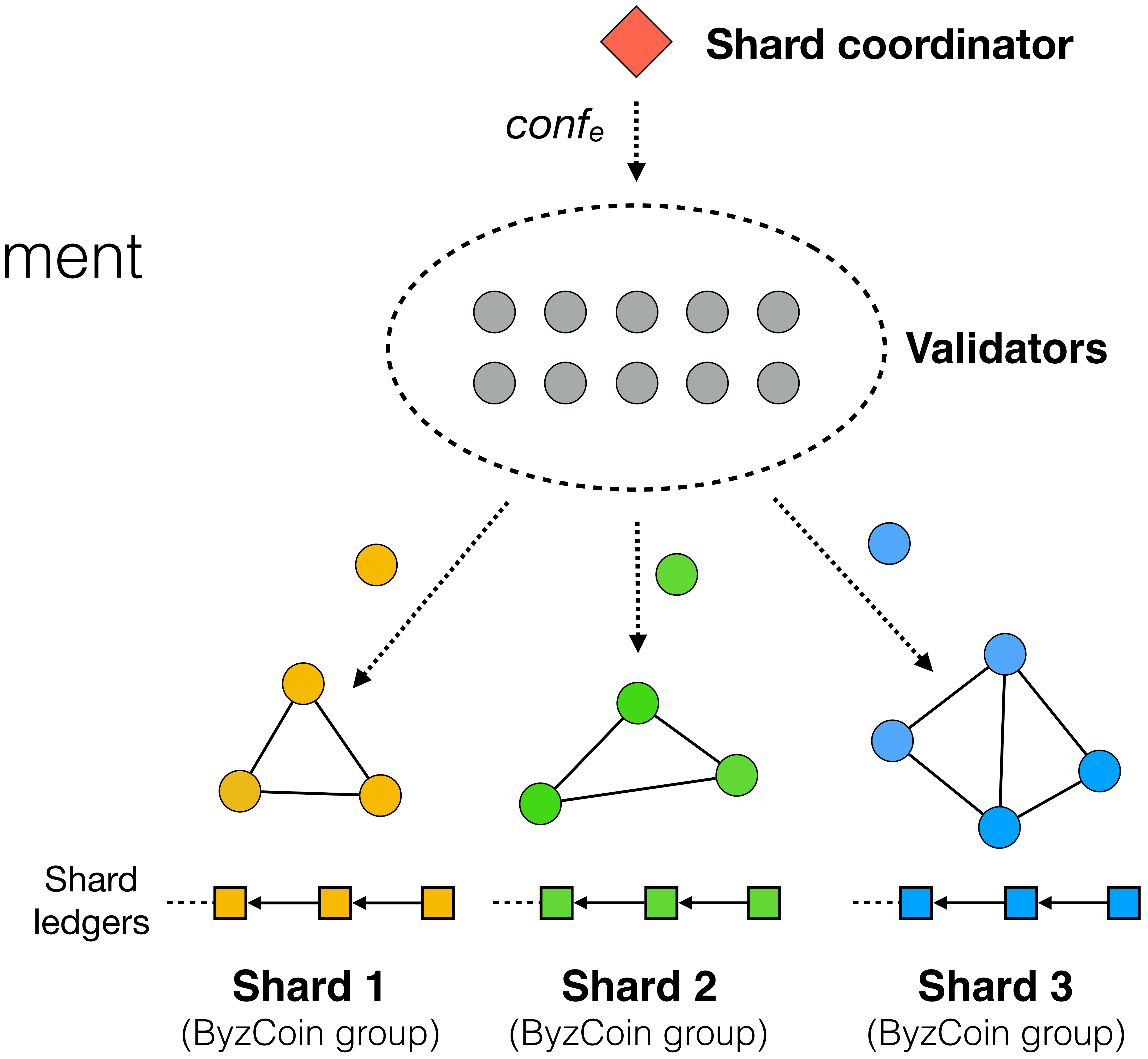
Strawman: SimpleLedger

Security Drawbacks

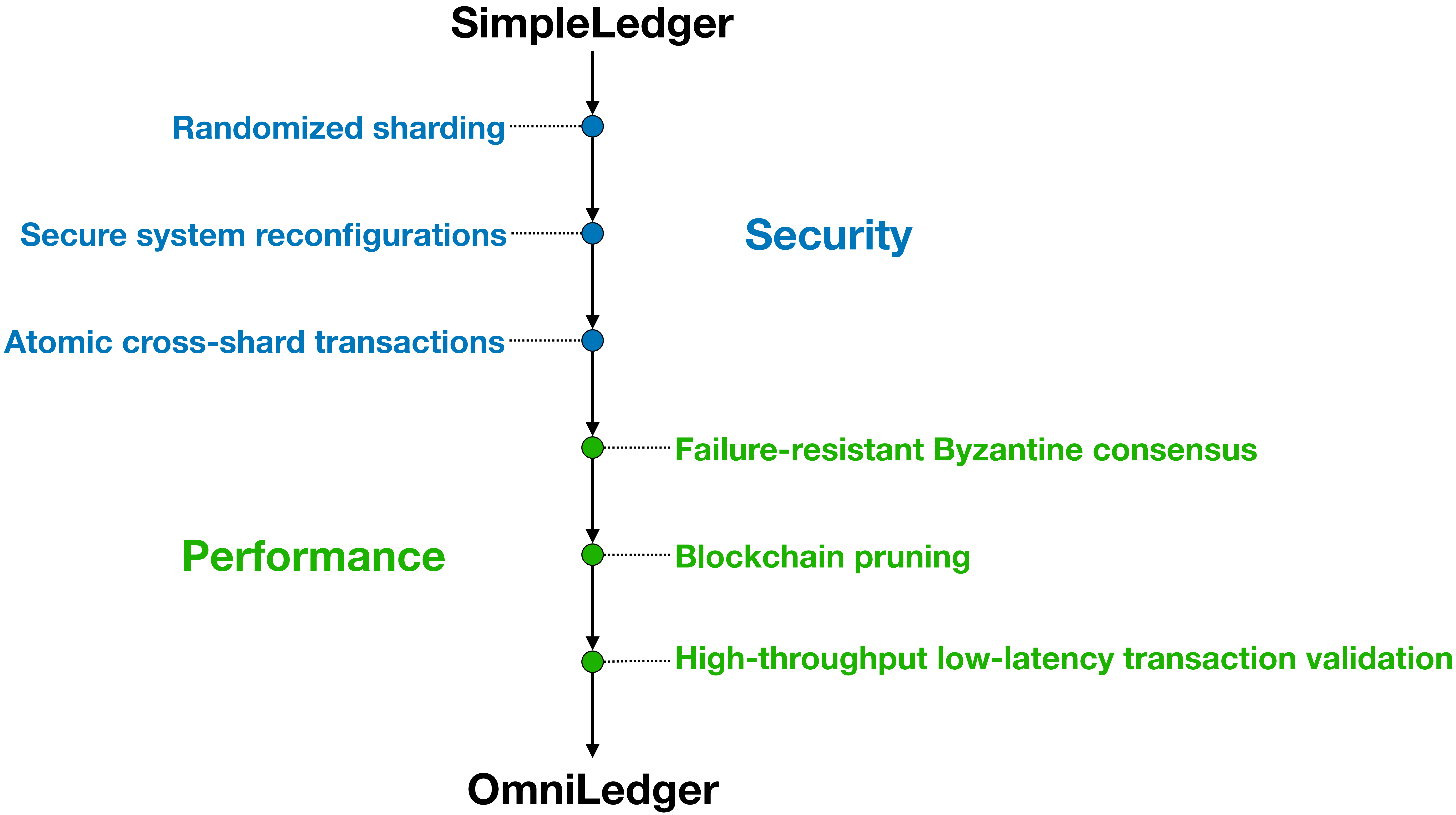
- Shard coordinator: trusted third party
- No tx processing during validator re-assignment
- No cross-shard tx support

Performance Drawbacks

- ByzCoin failure mode
- High storage and bootstrapping cost
- Throughput vs. latency trade-off



Roadmap



SimpleLedger

Randomized sharding

Secure system reconfigurations

Atomic cross-shard transactions

Security

Failure-resistant Byzantine consensus

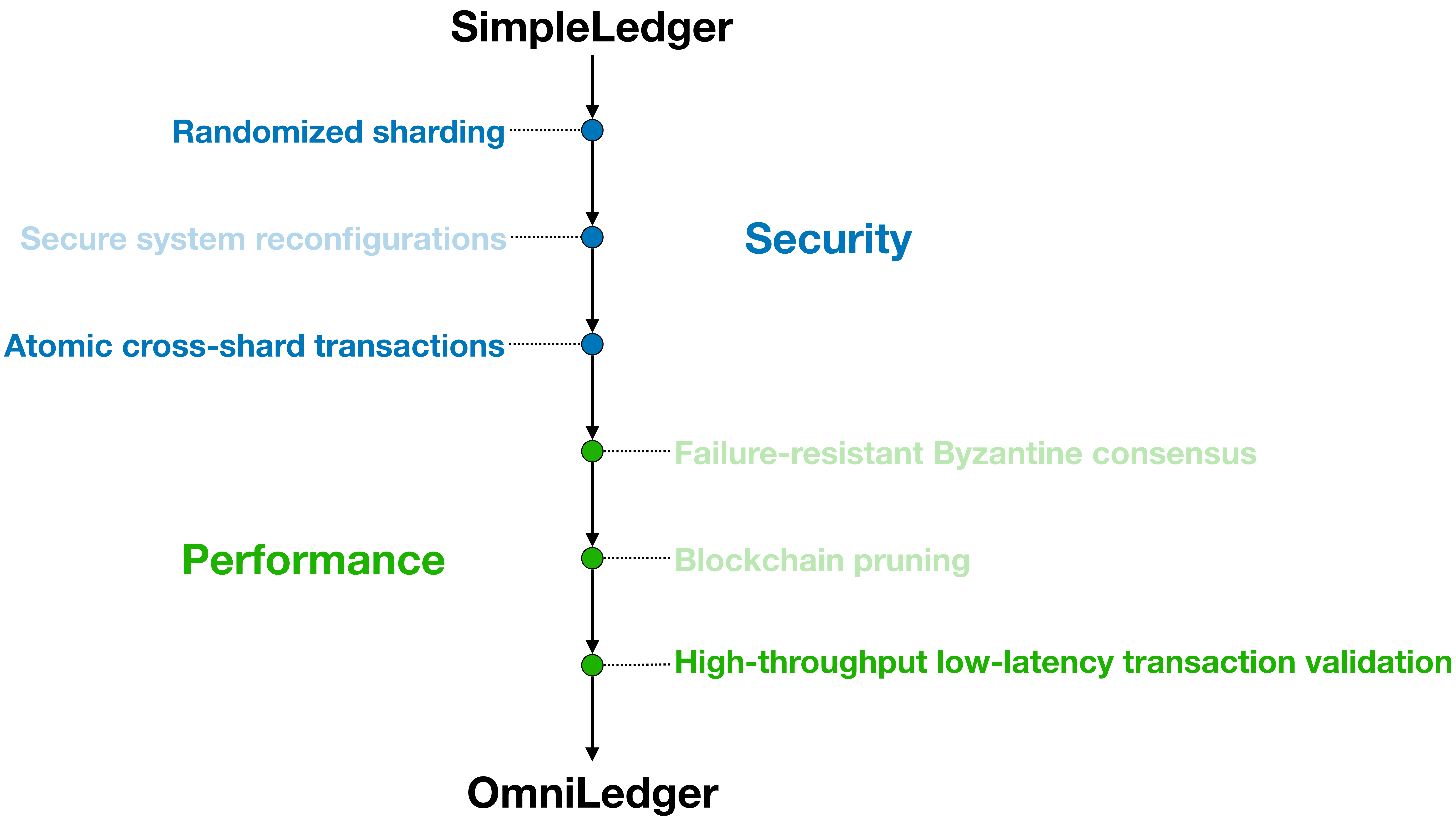
Blockchain pruning

High-throughput low-latency transaction validation

Performance

OmniLedger

Roadmap



SimpleLedger

Randomized sharding

Secure system reconfigurations

Atomic cross-shard transactions

Security

Failure-resistant Byzantine consensus

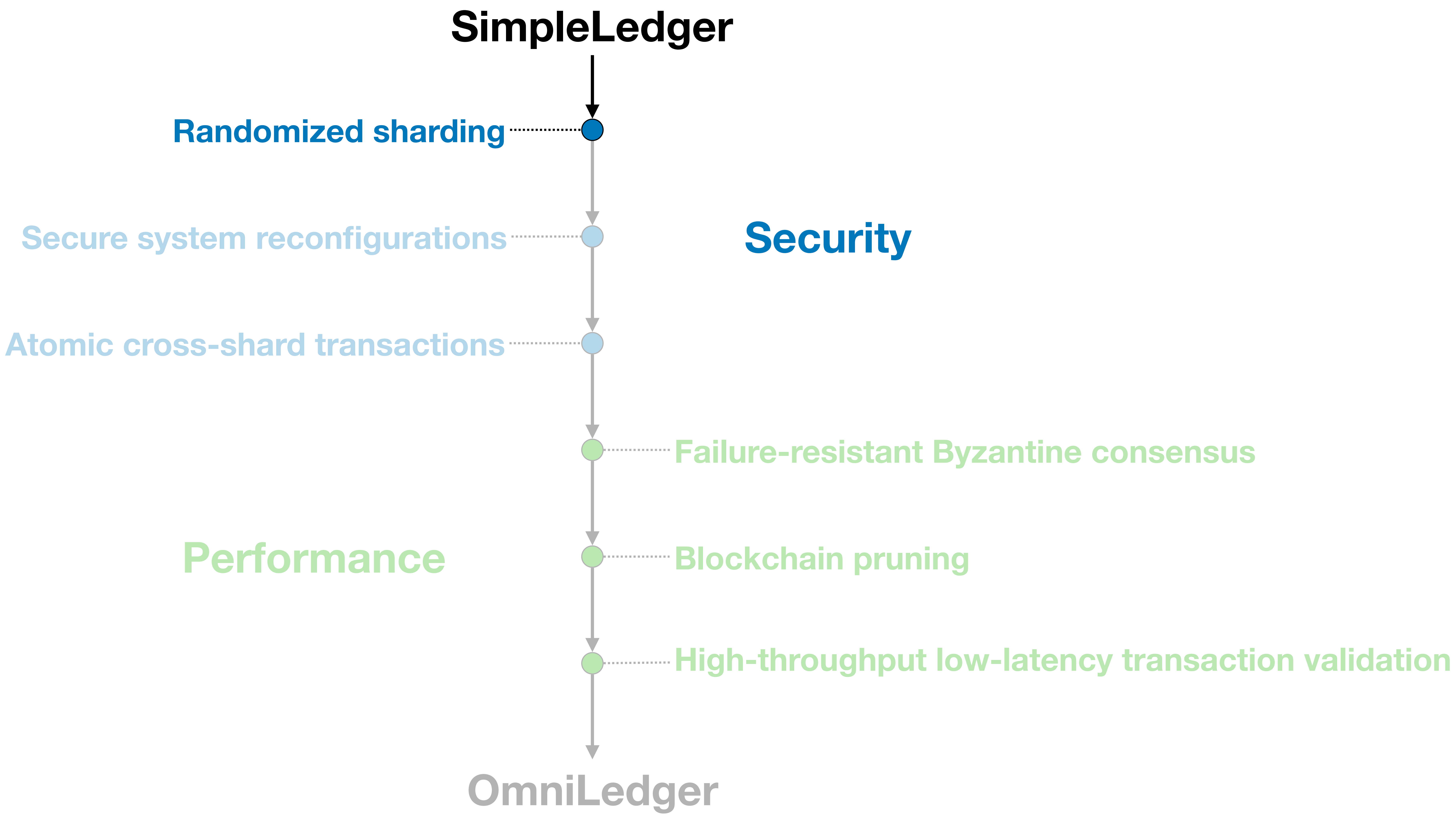
Blockchain pruning

High-throughput low-latency transaction validation

Performance

OmniLedger

Roadmap



SimpleLedger

Randomized sharding

Secure system reconfigurations

Security

Atomic cross-shard transactions

Failure-resistant Byzantine consensus

Performance

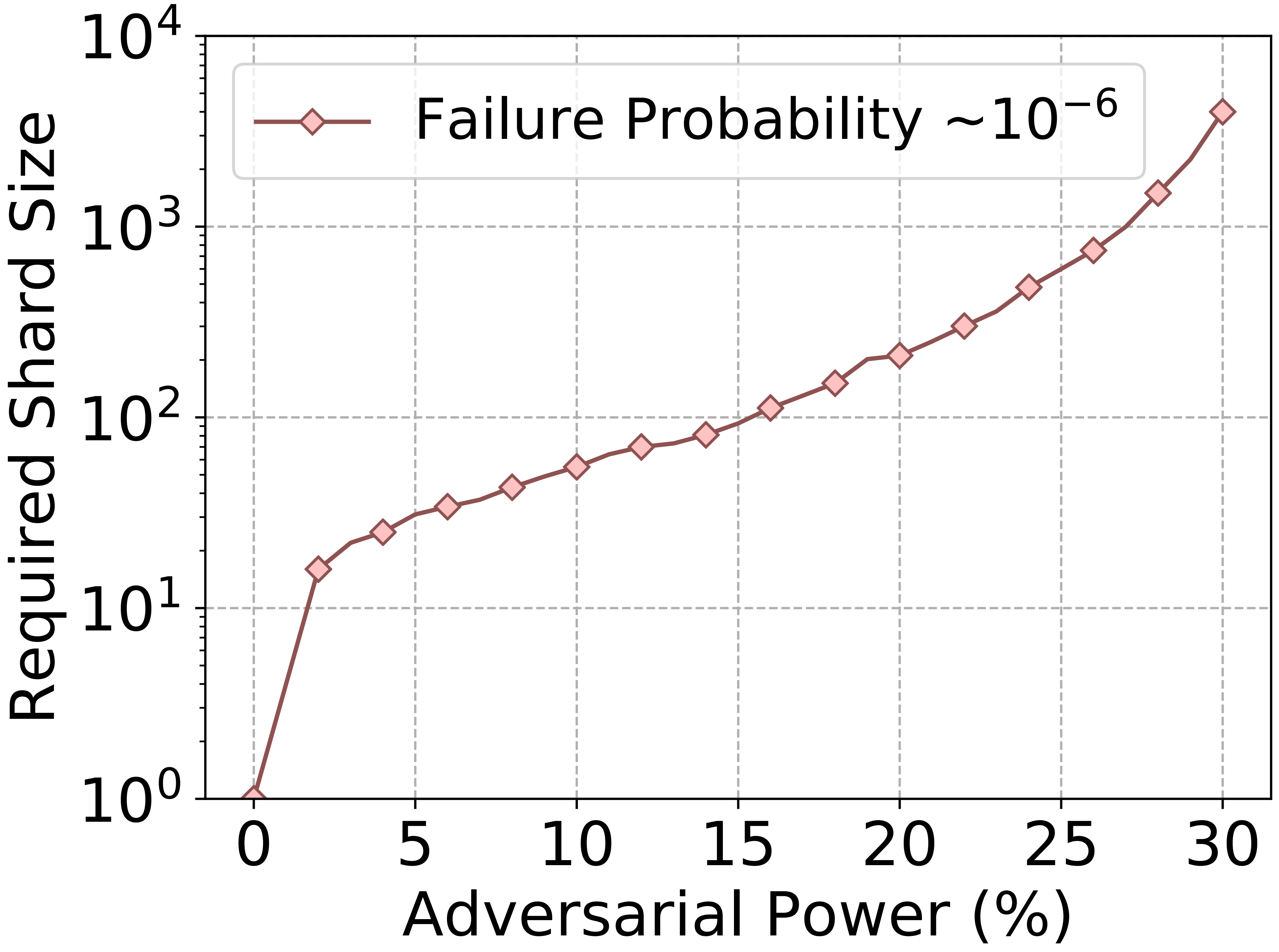
Blockchain pruning

High-throughput low-latency transaction validation

OmniLedger

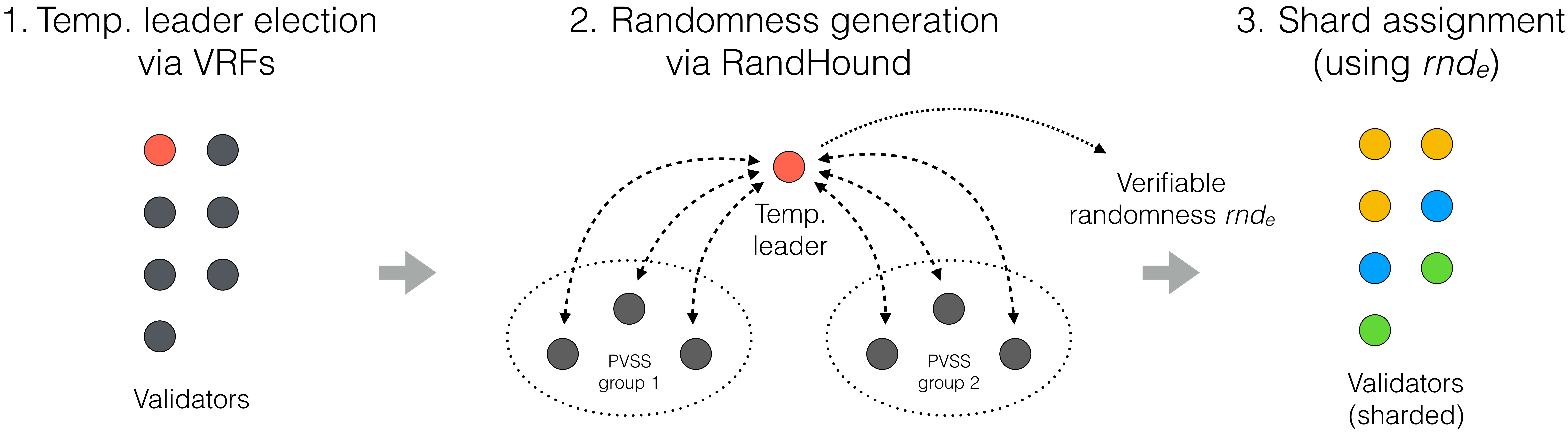
Shard Validator Assignment

- **How to assign validators to shards?**
  - Deterministically: Adversary can use predictable assignments to his advantage ⛔
  - Randomly: Adversary cannot control or predict assignment ✅
- **How to ensure long-term shard security against an adaptive adversary?**
  - Make shards large enough
  - Periodically re-assign validators to shards



Shard Validator Assignment

- **Challenge:** Unbiasable, unpredictable and scalable shard validator assignment
- **Solution:** Combine VRF-based lottery and unbiasable randomness protocol for sharding

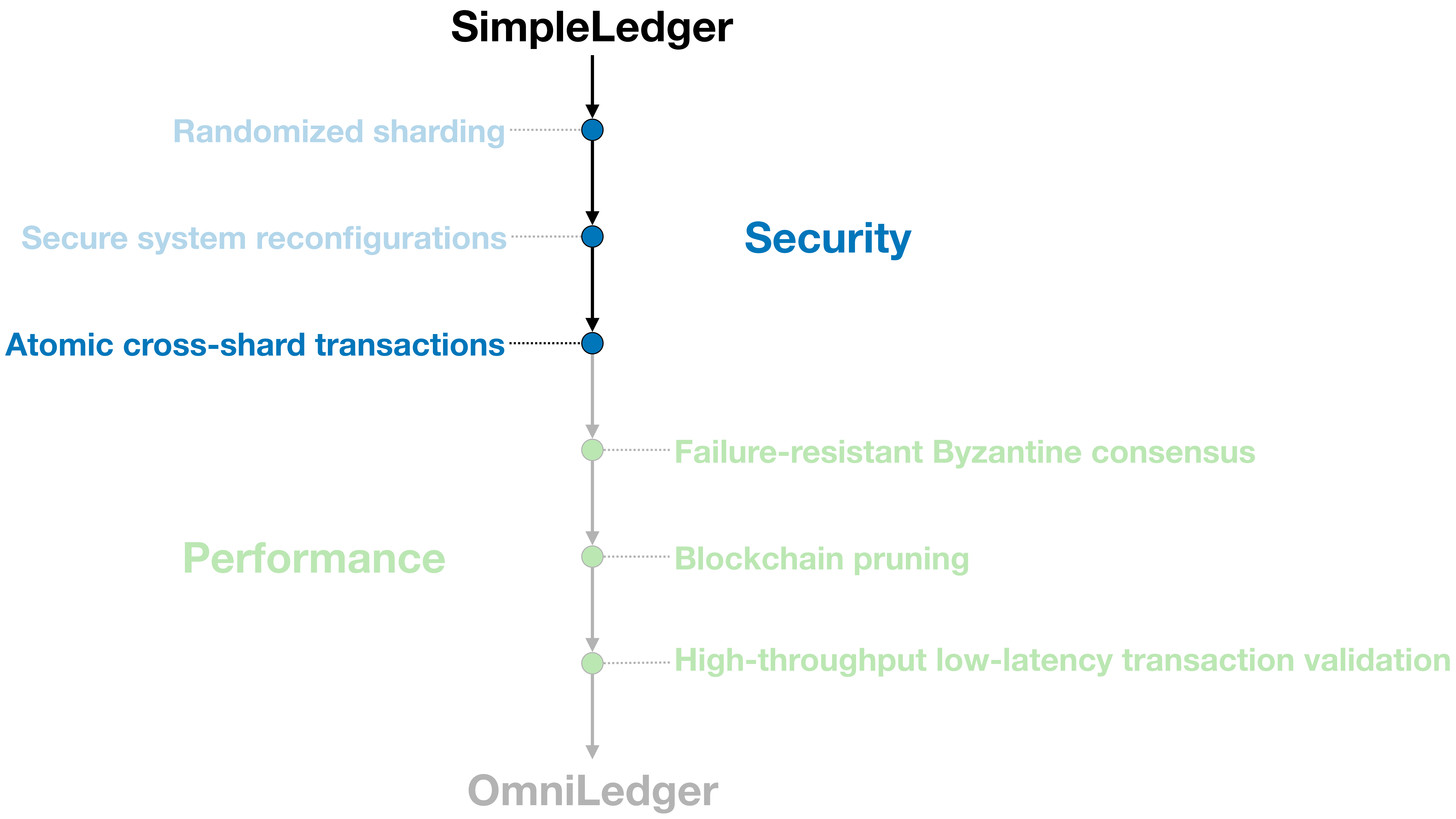


1. Temp. leader election via VRFs

2. Randomness generation via RandHound

3. Shard assignment (using rnd_e)

Roadmap



SimpleLedger

Randomized sharding

Secure system reconfigurations

Atomic cross-shard transactions

Security

Failure-resistant Byzantine consensus

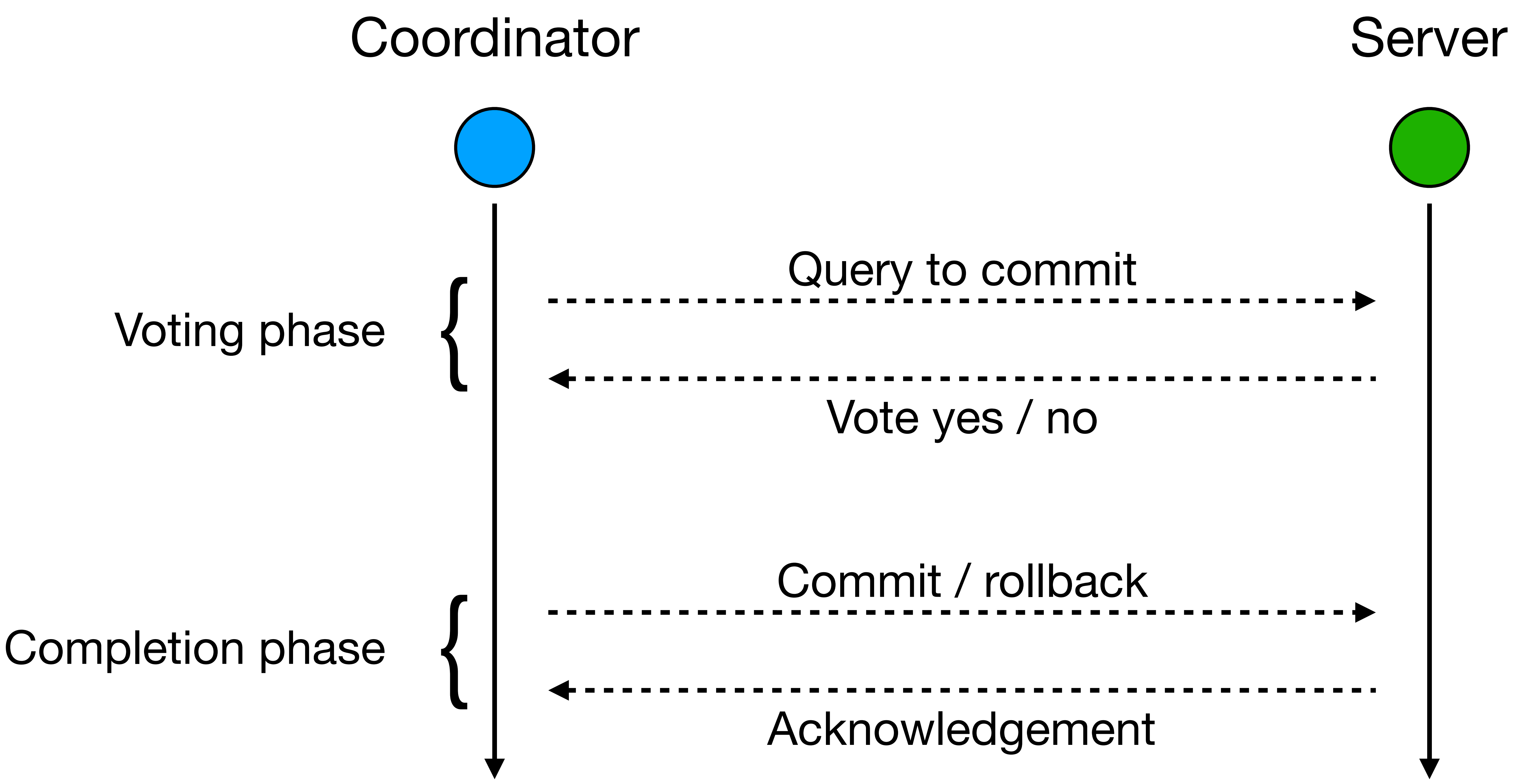
Performance

Blockchain pruning

High-throughput low-latency transaction validation

OmniLedger

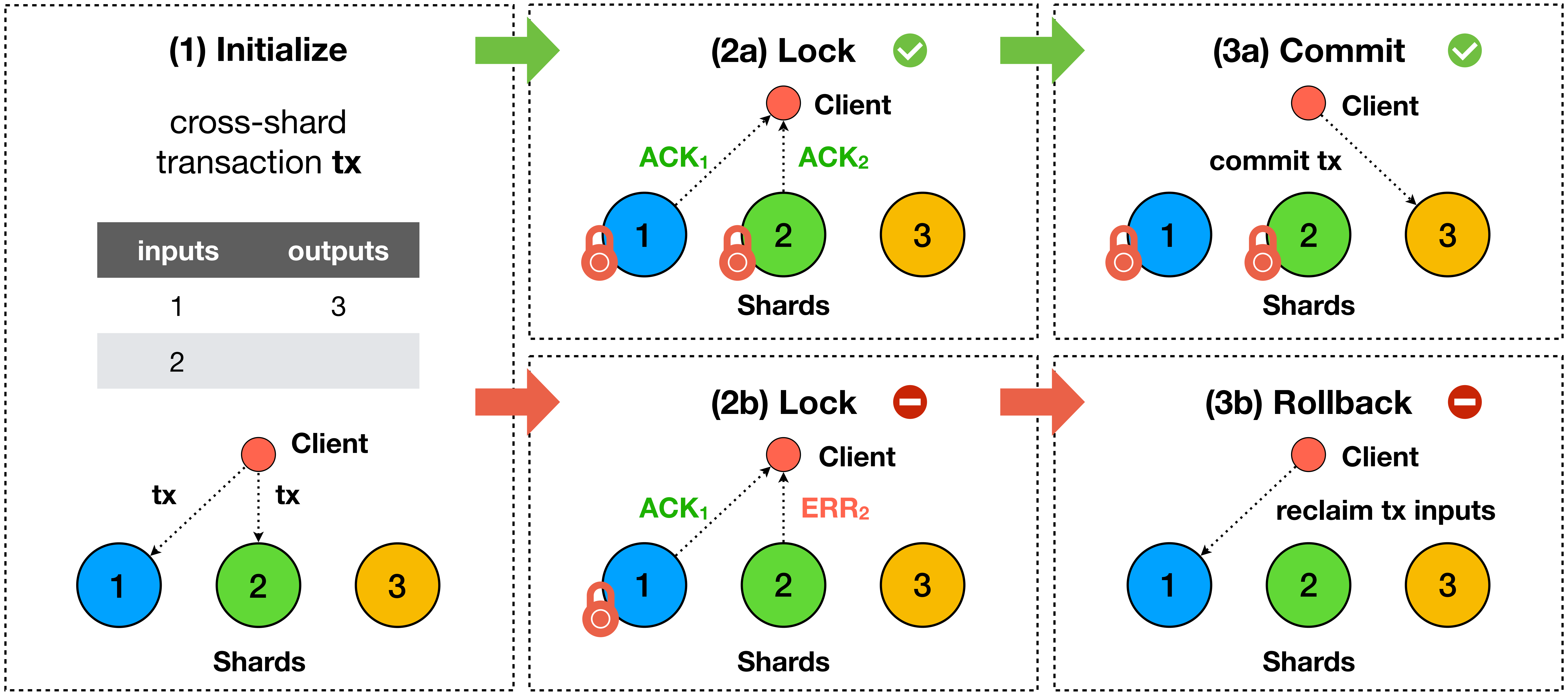
Two-Phase Commits



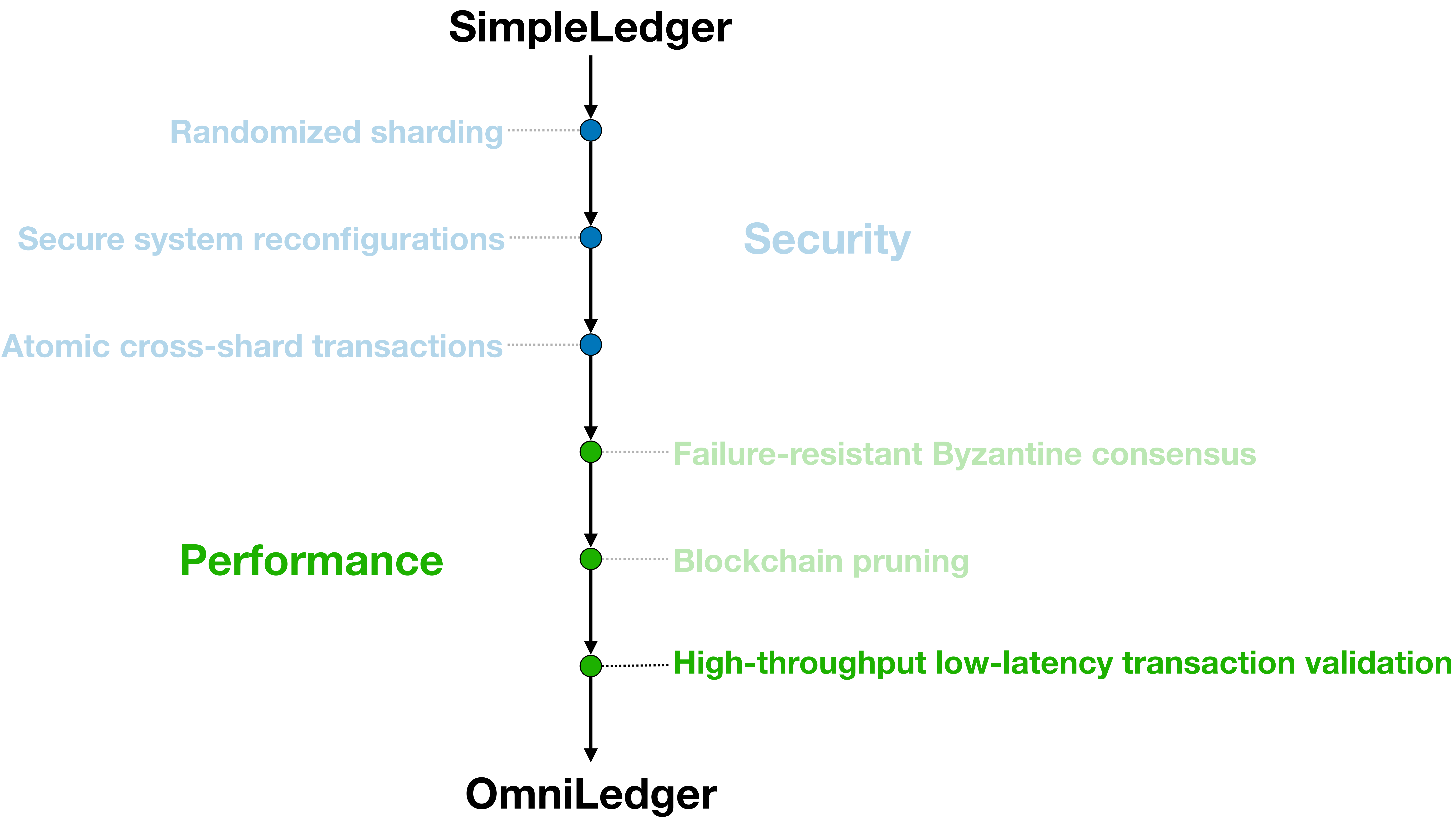
Problem: Does not work in a Byzantine setting as malicious nodes can always abort.

Atomix: Secure Cross-Shard Transactions

- **Challenge:** Cross-shard transactions commit atomically or abort eventually
- **Solution:** Atomix, a secure cross-shard transaction protocol (utilizing secure BFT shards)



Roadmap



SimpleLedger

Randomized sharding

Secure system reconfigurations

Security

Atomic cross-shard transactions

Failure-resistant Byzantine consensus

Performance

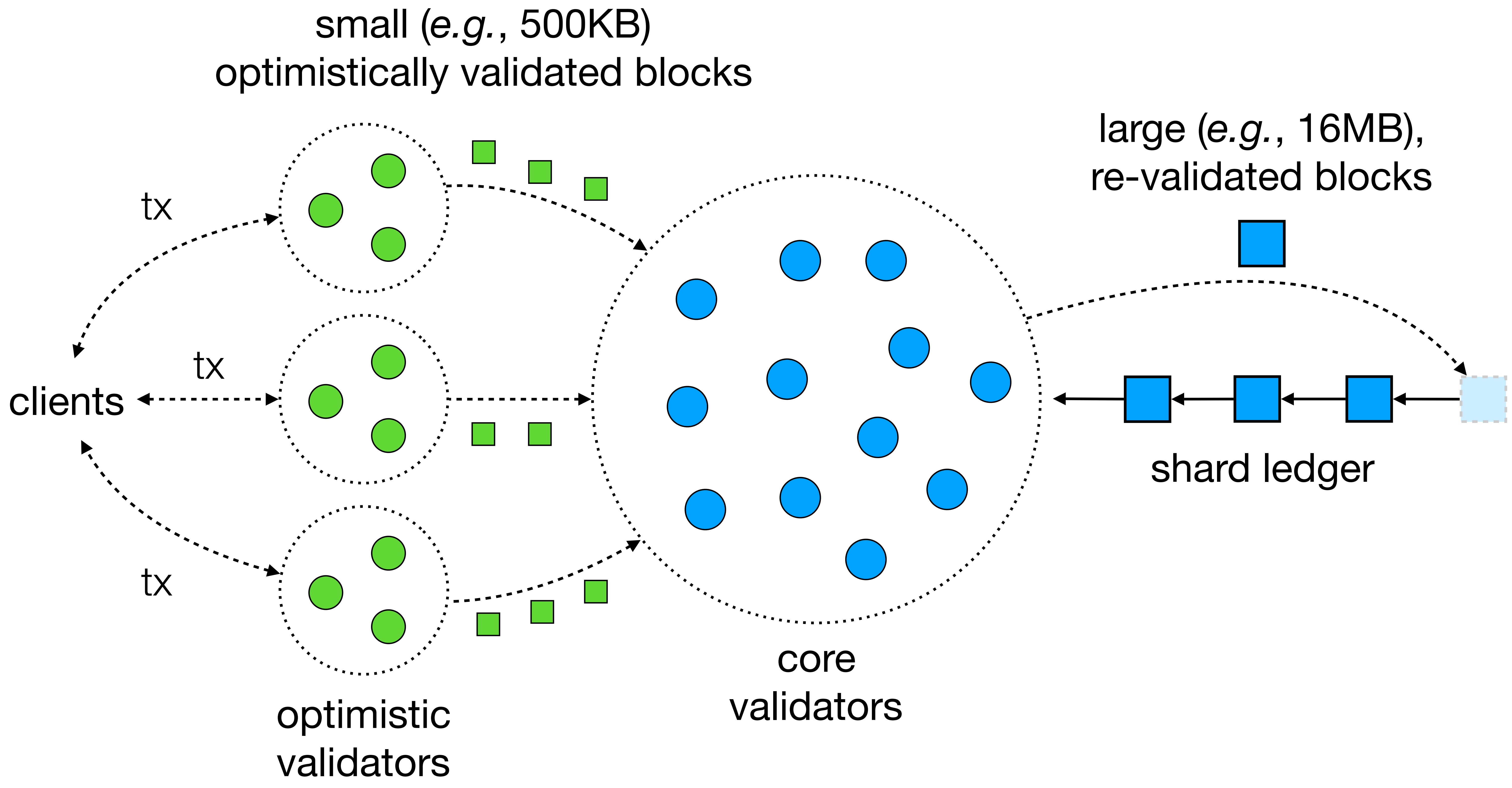
Blockchain pruning

High-throughput low-latency transaction validation

OmniLedger

Trust-but-Verify Transaction Validation

- **Challenge:** Latency vs. throughput trade-off
- **Solution:** Two-level “trust-but-verify” validation to get low latency *and* high throughput



Talk Outline

- Motivation
- OmniLedger
- **Evaluation**
- Conclusion

Implementation & Experimental Setup

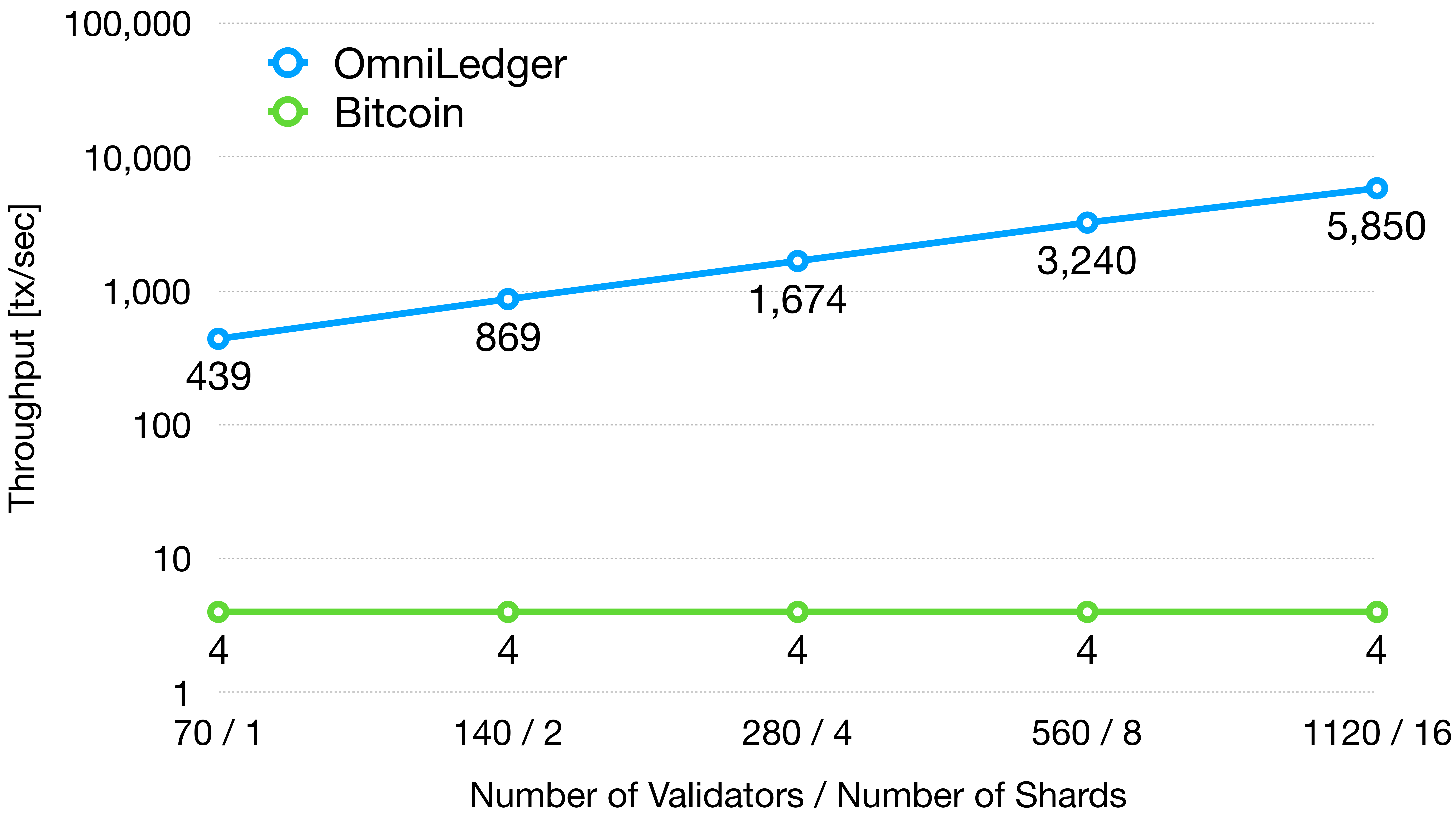
Implementation

- Go versions of OmniLedger and its subprotocols (ByzCoinX, Atomix, etc.)
- Based on DEDIS code
  - Kyber crypto library
  - Onet network library
  - Cothority framework
- https://github.com/dedis

DeterLab Setup

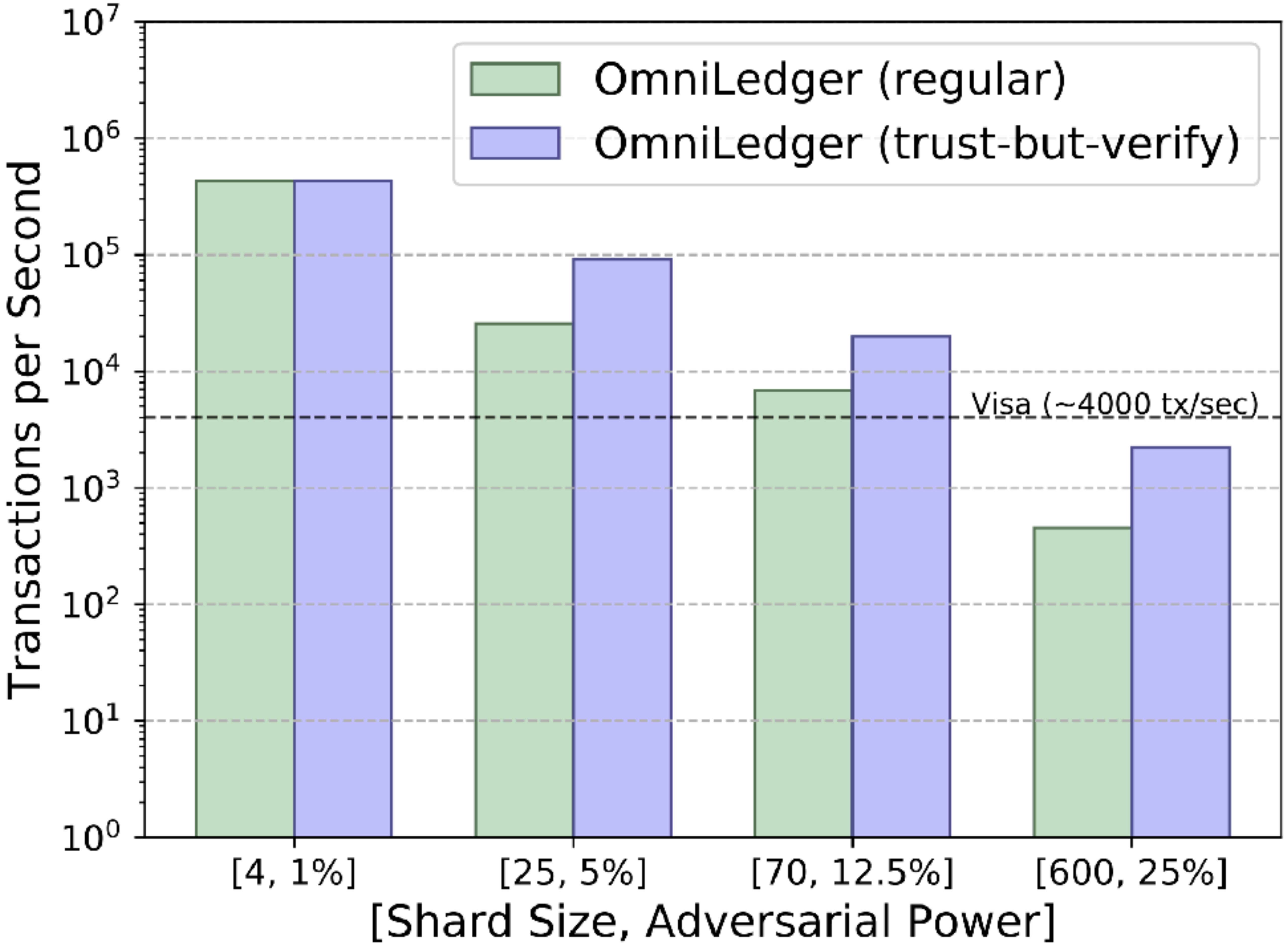
- 48 physical machines
  - Intel Xeon E5-2420 v2 (6 cores @ 2.2 GHz)
  - 24 GB RAM
  - 10 Gbps network link
- Realistic network configurations
  - 20 Mbps bandwidth
  - 200 ms round-trip latency

Evaluation: Scale-Out



For a 12.5%-adversary

Evaluation: Maximum Throughput



Results for 1800 validators

Evaluation: Latency

Transaction confirmation latency in *seconds* for regular and mutli-level validation

#shards, adversary	**4, 1%**	**25, 5%**	**70, 12.5%**	**600, 25%**	
OmniLedger regular	1.38	5.99	8.04	14.52	1 MB blocks
OmniLedger confirmation	1.38	1.38	1.38	4.48	500 KB blocks
OmniLedger consistency	1.38	55.89	41.89	62.96	16 MB blocks
Bitcoin confirmation	600	600	600	600	1 MB blocks
Bitcoin consistency	3600	3600	3600	3600	1 MB blocks

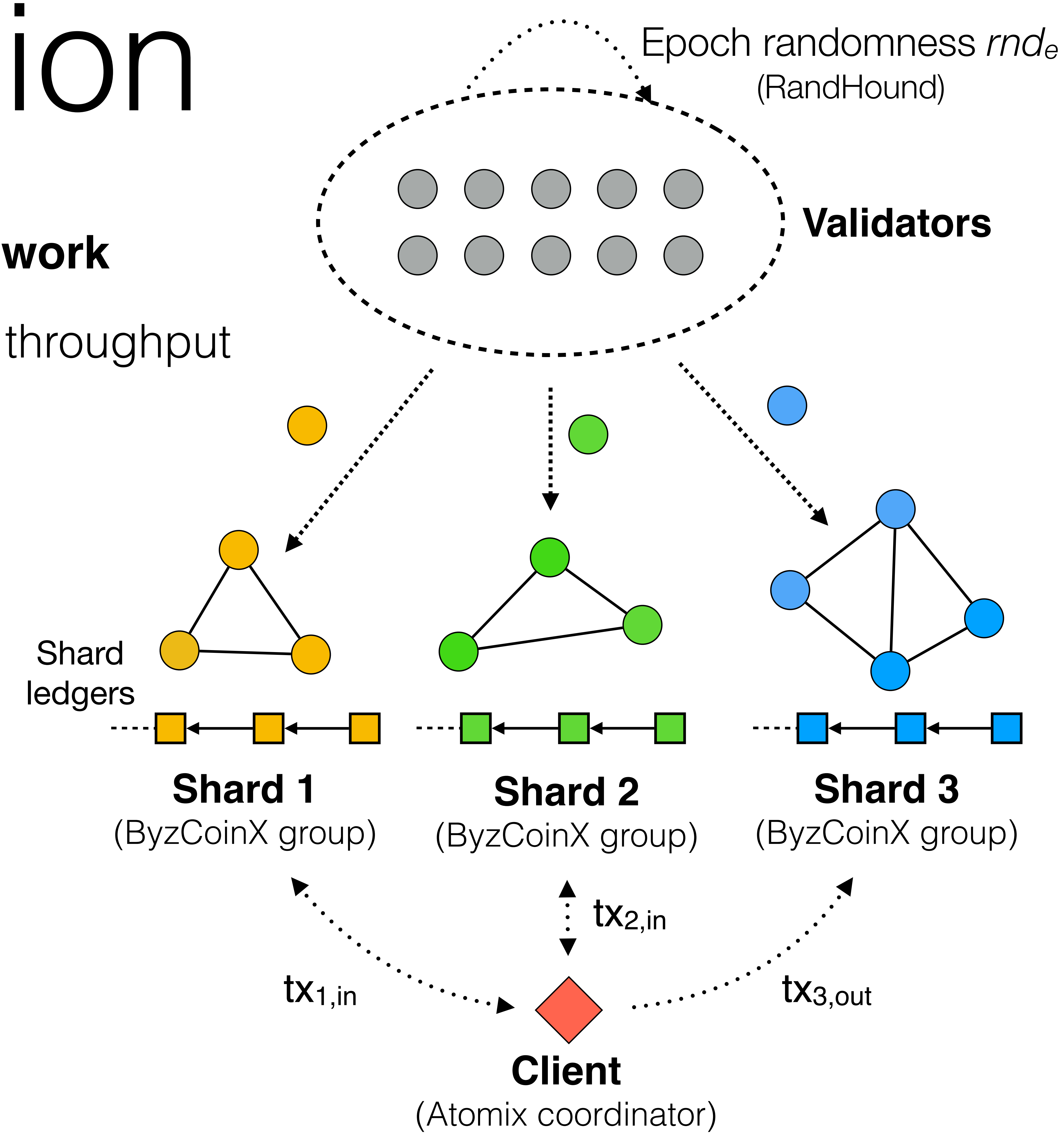
latency increase since optimistically validated blocks are batched into larger blocks for final validation to get better throughput

Talk Outline

- Motivation
- OmniLedger
- Experimental Results
- **Conclusion**

Conclusion

- **OmniLedger – Secure scale-out distributed ledger framework**
  - Sharding via unbiasable randomness for linearly-scaling throughput
  - Atomix: Client-managed cross-shard transactions
  - ByzCoinX: Robust intra-shard BFT consensus
  - Trust-but-verify validation for low latency *and* high throughput
  - For PoW, PoS, permissioned, etc.
- **Paper:** ia.cr/2017/406 (published at IEEE S&P'18)
- **Code:** https://github.com/dedis



Thank you!

Questions?

Ewa Syta
ewa.syta@trincoll.edu