MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

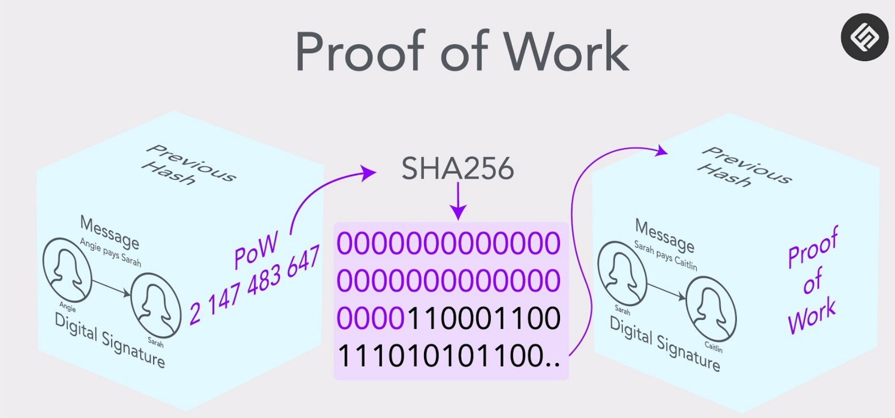
# Algorand: Scaling Byzantine Agreements for Cryptocurrencies

**Authors: Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, Nickolai Zeldovich**
**Presenter: Ximin Lin**

# Background

**Proof of Work:**

1. Assumption: honest people have more computation power than bad people
2. Compute nounce number in the next block to make hash of the next block start with some length of 0.




https://www.coinkolik.com/wp-content/uploads/2021/03/Proof-of-work-no-purple-line.jpg

# Background

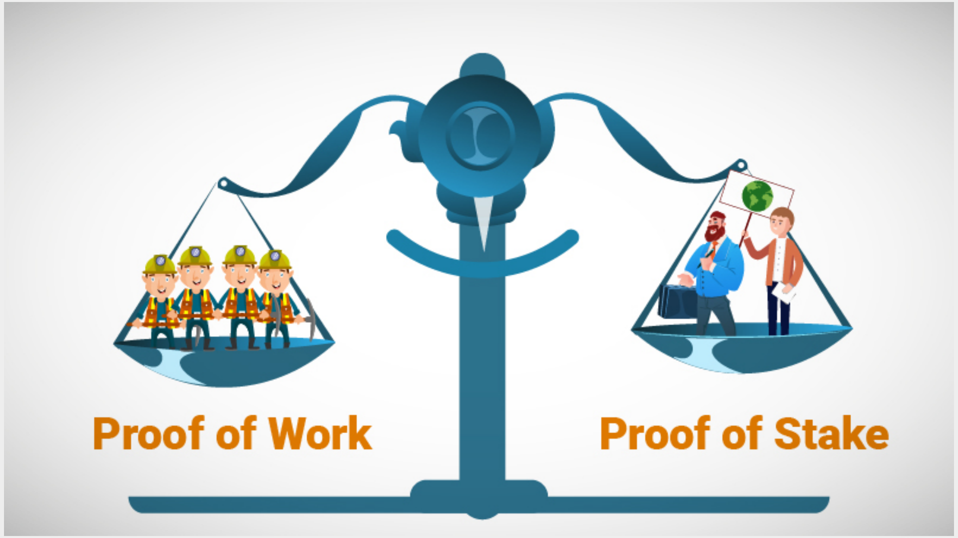
**Byzantine Consensus:**

1. PBFT: Sybil Attacks
2. BFT-2F: Forking-consensus with only over ½ of honest servers
3. HoneyBadger: Centralized
4. Bitcoin-NG, Hybrid-Consensus: Forks
5. Stellar: complex trust structure and assumptions

# Background

**Proof of Stake:**

1. Honest people have more money than bad people
2. People with more money tend to be chosen as proposers or acceptors for creation of the next blocks



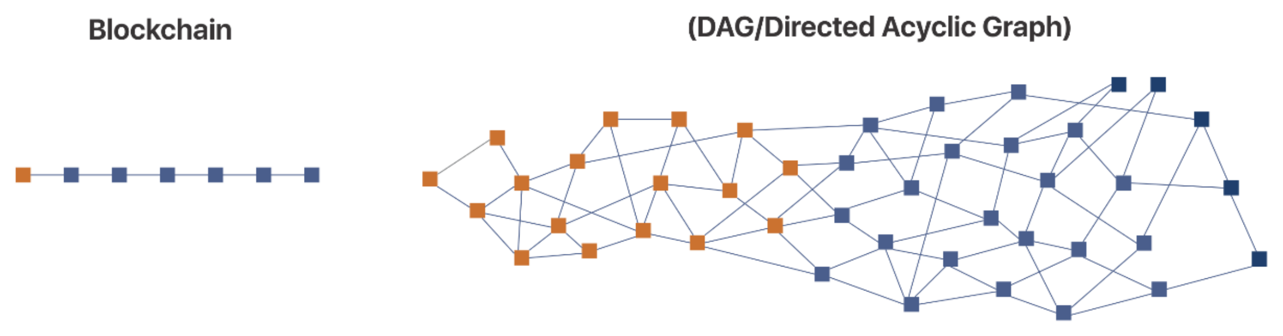

https://coindoo.com/wp-content/uploads/2019/03/Proof-of-Work-vs.-Proof-of-Stake.jpg

MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

# Background

## Trees and DAGs:

Increase bitcoin throughput by replacing chained structure ledger with tree or directed acyclic graph.




https://www.cbcamerica.org/Content/Images/dlt.png

# Motivation

1. PoW is slow: (eg. Bitcoin: 10 mins for one block generation, 6 blocks to secure a transaction, in total 1 hour to confirm a transaction)

2. Fork is slow (Need to wait for more blocks to confirm a branch)

# Challenges

1. Sybil Attack ← PoS
2. Scale to millions of users ← Consensus by Committee
3. Resilient to DoS Attack ← Cryptographic Sortition, Participant Replacement

# Goals & Assumptions

1. Safety: If one honest user accepts transaction A, all other honest user accepts A.

2. Liveness: make progress

3. Assumption:
    1. Honest users hold more than 2/3 of total wealth of chain
    2. Strong synchrony for liveness
    3. "weak/partial synchrony" for safety: for every period of length b, there must be a strongly synchronous period s < b
    4. loosely synchronous clocks among users

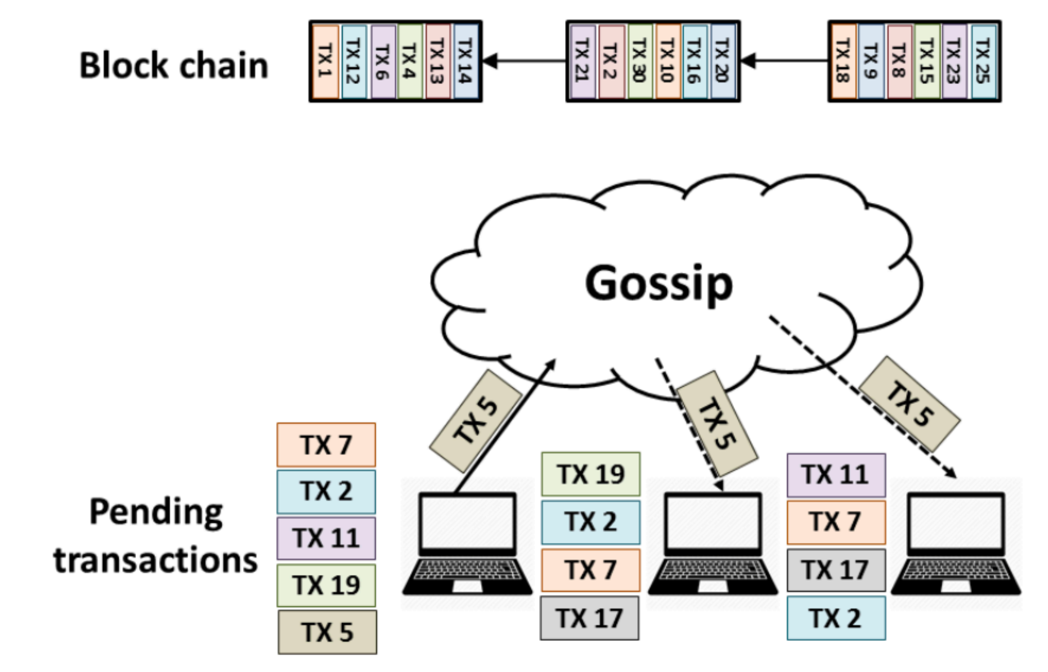
# Overview

1. Gossip Protocol
2. Cryptographic Sortition to select committee
3. Block Proposal
4. BA*: Byzantine Agreement Protocol: tentative consensus; final consensus
5. Efficiency

# Gossip Protocol

Every user *"gossip"* the transactions to some other users

To avoid forwarding loop, each user does not relay the same transaction twice.



# Cryptographic Sortition

**Cryptographic Sortition:**

Algorithm for choosing a random subset of all users to form:

1. Proposers
2. Consensus Committee

**Idea**: the probability of selecting a user is proportional to the money it has

# Cryptographic Sortition

**VRF(verifiable random function):**

Input: data x, and a secret key

Output: hash, and proof

Hash appears random to anybody who does not know secret key
Proof enables anybody who knows the public key to verify that the hash corresponds to data x

Can be used to generate hash as random number if provided a random seed.

# Cryptographic Sortition

## Selection Procedure:

Consider one unit of Algorand as a ”sub-user”.

Total amount of currency is W

Each sub-user is selected with probability p = t/W (t controls the number of selected users)

Each user can be selected multiple times

# Cryptographic Sortition

**Selection Procedure:**

**procedure** Sortition($sk, seed, \tau, role, w, W$):

$\langle hash, \pi \rangle \leftarrow \text{VRF}_{sk}(seed || role)$

$p \leftarrow \frac{\tau}{W}$

$j \leftarrow 0$

**while** $\frac{hash}{2^{hashlen}} \notin \left[ \sum_{k=0}^{j} B(k; w, p), \sum_{k=0}^{j+1} B(k; w, p) \right)$ **do**

  $j{+}{+}$

**return** $\langle hash, \pi, j \rangle$

**Algorithm 1:** The cryptographic sortition algorithm.

**procedure** VerifySort($pk, hash, \pi, seed, \tau, role, w, W$):

**if** $\neg VerifyVRF_{pk}(hash, \pi, seed || role)$ **then return** 0;

$p \leftarrow \frac{\tau}{W}$

$j \leftarrow 0$

**while** $\frac{hash}{2^{hashlen}} \notin \left[ \sum_{k=0}^{j} B(k; w, p), \sum_{k=0}^{j+1} B(k; w, p) \right)$ **do**

  $j{+}{+}$

**return** j

**Algorithm 2:** Pseudocode for verifying sortition of a user with public key *pk*.

B() computes the probability that k algos are selected from total w algos.

Consider $\frac{hash}{2^{hashlen}}$ as random number from [0,1].

j represents the number of times a user is selected.

# Cryptographic Sortition

**Choosing the seed:**

1. Algorand requires a publicly known seed for everyone to use for VRF

2. Cannot be known in advance or controlled by anyone

3. $seed_r = \mathrm{H}(\mathrm{seed}_{\mathrm{r}-1}||\mathrm{r})$. H is a cryptographic hash function

4. Refreshed every R rounds, $seed_r = \mathrm{H}(\mathrm{seed}_{\mathrm{r}-1-(r \bmod R)}||\mathrm{r})$

# Cryptographic Sortition

**Cryptographic Sortition:**

Algorithm for choosing a random subset of all users to form:

1. Proposers
2. Consensus Committee

# Block Proposal

**Minimize unnecessary block transmissions:**

1. During selection process, **priority** is also assigned to selected proposers
2. Users only accept blocks with the highest priority
3. Block metadata (priority, timestamp, proof…) has size ~200 Bytes
4. Whole Block (mostly transactions) has size ~ 1MB

# Block Proposal

**Timeout for block proposal:**

1. Does not affect safety but important for performance
2. If timeout, accept an empty block.
3. If enough user timeout, consensus on empty block will be reached.
4. Timeout value is calculated considering $\lambda_{stepvar} + \lambda_{priority}$

# Block Proposal

**Malicious Proposers**

1. Will try to propose different blocks for different acceptors
2. Happen in low probability

MICHIGAN ENGINEERING
UNIVERSITY OF MICHIGAN

# BA*

**Byzantine Agreement Protocol**

1. First phase, every honest user agrees on either empty block or the same non-empty block.
2. Second phase, every honest user agrees on the same block.
3. In each step, every committee member vote and count votes. Users receiving more than a threshold of votes for some value will vote for this value in next step.
4. If committee member timeouts on insufficient task, will decide what value to vote next by the step number.

# BA*

## Overview Procedure

**procedure** $BA\star(ctx, round, block)$:

$hblock \leftarrow \text{Reduction}(ctx, round, H(block))$

$hblock_\star \leftarrow \text{Binary}BA\star(ctx, round, hblock)$

// Check if we reached "final" or "tentative" consensus

$r \leftarrow \text{CountVotes}(ctx, round, \text{FINAL}, T_{\text{FINAL}}, \tau_{\text{FINAL}}, \lambda_{\text{STEP}})$

**if** $hblock_\star = r$ **then**

  **return** $\langle \text{FINAL}, \text{BlockOfHash}(hblock_\star) \rangle$

**else**

  **return** $\langle \text{TENTATIVE}, \text{BlockOfHash}(hblock_\star) \rangle$

**Algorithm 3:** Running *BA★* for the next *round*, with a proposed *block*. H is a cryptographic hash function.

# BA*

**Overview Procedure**

**procedure** $BA\star(ctx, round, block)$:

$hblock \leftarrow \text{Reduction}(ctx, round, H(block))$

$hblock_\star \leftarrow \text{Binary}BA\star(ctx, round, hblock)$

// Check if we reached “final” or “tentative” consensus

$r \leftarrow \text{CountVotes}(ctx, round, \text{FINAL}, T_{\text{FINAL}}, \tau_{\text{FINAL}}, \lambda_{\text{STEP}})$

**if** $hblock_\star = r$ **then**

  **return** $\langle \text{FINAL}, \text{BlockOfHash}(hblock_\star) \rangle$

**else**

  **return** $\langle \text{TENTATIVE}, \text{BlockOfHash}(hblock_\star) \rangle$

**Algorithm 3:** Running *BA⋆* for the next *round*, with a proposed *block*. H is a cryptographic hash function.

For efficiency, *BA⋆* votes for hashes of blocks, instead of entire block contents. The *BA⋆* also determines whether it established final or tentative consensus.

# BA*

## Voting

**procedure** CommitteeVote($ctx, round, step, \tau, value$):

// check if user is in committee using Sortition (Alg. 1)

$role \leftarrow \langle$"committee"$, round, step\rangle$

$\langle sorthash, \pi, j\rangle \leftarrow$ Sortition$(user.sk, ctx.seed, \tau, role, ctx.weight[user.pk], ctx.W)$

// only committee members originate a message

**if** $j > 0$ **then**

  Gossip$(\langle user.pk,$ Signed$_{user.sk}(round, step, sorthash, \pi, H(ctx.last_block), value)\rangle)$

**Algorithm 4:** Voting for *value* by committee members. *user.sk* and *user.pk* are the user's private and public keys.

# BA*

## Counting Votes

```
procedure CountVotes(ctx, round, step, T, τ, λ):
start ← Time()
counts ← {}     // hash table, new keys mapped to 0
voters ← {}
msgs ← incomingMsgs[round, step].iterator()
while TRUE do
    m ← msgs.next()
    if m = ⊥ then
        if Time() > start + λ then return TIMEOUT;
    else
        ⟨votes, value, sorthash⟩ ← ProcessMsg(ctx, τ, m)
        if pk ∈ voters or votes = 0 then continue;
        voters ∪ = {pk}
        counts[value] + = votes
        // if we got enough votes, then output this value
        if counts[value] > T · τ then
            return value
```

**Algorithm 5:** Counting votes for *round* and *step*.

```
procedure ProcessMsg(ctx, τ, m):
⟨pk, signed_m⟩ ← m
if VerifySignature(pk, signed_m) ≠ OK then
    return ⟨0, ⊥, ⊥⟩
⟨round, step, sorthash, π, hprev, value⟩ ← signed_m
// discard messages that do not extend this chain
if hprev ≠ H(ctx.last_block) then return ⟨0, ⊥, ⊥⟩;
votes ← VerifySort(pk, sorthash, π, ctx.seed, τ,
        ⟨"committee", round, step⟩, ctx.weight[pk], ctx.W)
return ⟨votes, value, sorthash⟩
```

**Algorithm 6:** Validating incoming vote message *m*.

# BA*

## Reduction

```
procedure Reduction(ctx, round, hblock):
// step 1: gossip the block hash
CommitteeVote(ctx, round, REDUCTION_ONE,
        τ_STEP, hblock)
// other users might still be waiting for block proposals,
// so set timeout for λ_BLOCK + λ_STEP
hblock_1 ← CountVotes(ctx, round, REDUCTION_ONE,
        T_STEP, τ_STEP, λ_BLOCK + λ_STEP)
// step 2: re-gossip the popular block hash
empty_hash ← H(Empty(round, H(ctx.last_block)))
if hblock_1 = TIMEOUT then
    CommitteeVote(ctx, round, REDUCTION_TWO,
            τ_STEP, empty_hash)
else
    CommitteeVote(ctx, round, REDUCTION_TWO,
            τ_STEP, hblock_1)
hblock_2 ← CountVotes(ctx, round, REDUCTION_TWO,
        T_STEP, τ_STEP, λ_STEP)
if hblock_2 = TIMEOUT then return empty_hash ;
else return hblock_2 ;
```

**Algorithm 7:** The two-step reduction.

# BA*

## Binary Agreement

```
procedure BinaryBA⋆(ctx, round, block_hash):
step ← 1
r ← block_hash
empty_hash ← H(Empty(round, H(ctx.last_block)))
while step < MaxSteps do
    CommitteeVote(ctx, round, step, τ_STEP, r)
    r ← CountVotes(ctx, round, step, T_STEP, τ_STEP, λ_STEP)
    if r = TIMEOUT then
        r ← block_hash
    else if r ≠ empty_hash then
        for step < s′ ≤ step + 3 do
            CommitteeVote(ctx, round, s′, τ_STEP, r)
        if step = 1 then
            CommitteeVote(ctx, round, FINAL, τ_FINAL, r)
        return r
    step++

    CommitteeVote(ctx, round, step, τ_STEP, r)
    r ← CountVotes(ctx, round, step, T_STEP, τ_STEP, λ_STEP)
    if r = TIMEOUT then
        r ← empty_hash
    else if r = empty_hash then
        for step < s′ ≤ step + 3 do
            CommitteeVote(ctx, round, s′, τ_STEP, r)
        return r
    step++

    CommitteeVote(ctx, round, step, τ_STEP, r)
    r ← CountVotes(ctx, round, step, T_STEP, τ_STEP, λ_STEP)
    if r = TIMEOUT then
        if CommonCoin(ctx, round, step, τ_STEP) = 0 then
            r ← block_hash
        else
            r ← empty_hash
    step++
// No consensus after MaxSteps; assume network
// problem, and rely on §8.2 to recover liveness.
HangForever()
```

**Algorithm 8:** Binary*BA*⋆ executes until consensus is reached on either *block_hash* or *empty_hash*.

# BA*

## Binary Agreement

1. Safety with strong synchrony

2. Safety with weak synchrony

# BA*

## Binary Agreement

```
procedure CommonCoin(ctx, round, step, τ):
minhash ← 2^hashlen
for m ∈ incomingMsgs[round, step] do
    ⟨votes, value, sorthash⟩ ← ProcessMsg(ctx, τ, m)
    for 1 ≤ j < votes do
        h ← H(sorthash||j)
        if h < minhash then minhash ← h;
return minhash mod 2
```

**Algorithm 9:** Computing a coin common to all users.

# Evaluation

## Latency

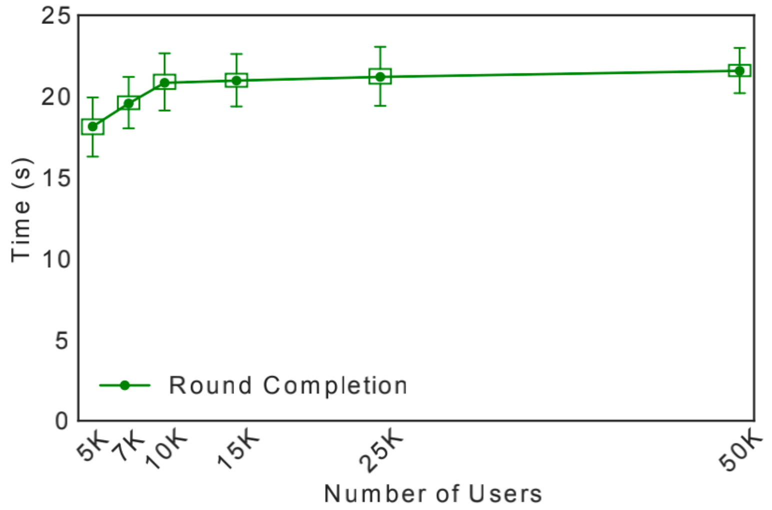



**Figure 5**: Latency for one round of Algorand, with 5,000 to 50,000 users.

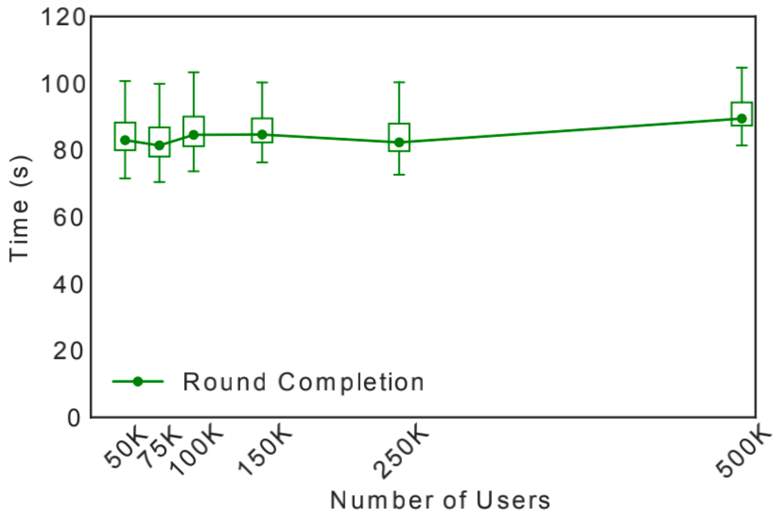



**Figure 6**: Latency for one round of Algorand in a configuration with 500 users per VM, using 100 to 1,000 VMs.

# Evaluation

## Block Size

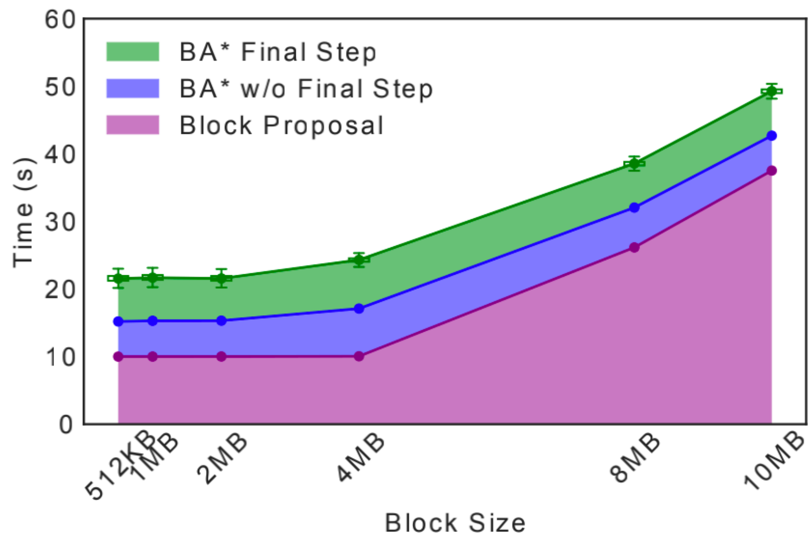



**Figure 7**: Latency for one round of Algorand as a function of the block size.

# Evaluation

## Malicious Users

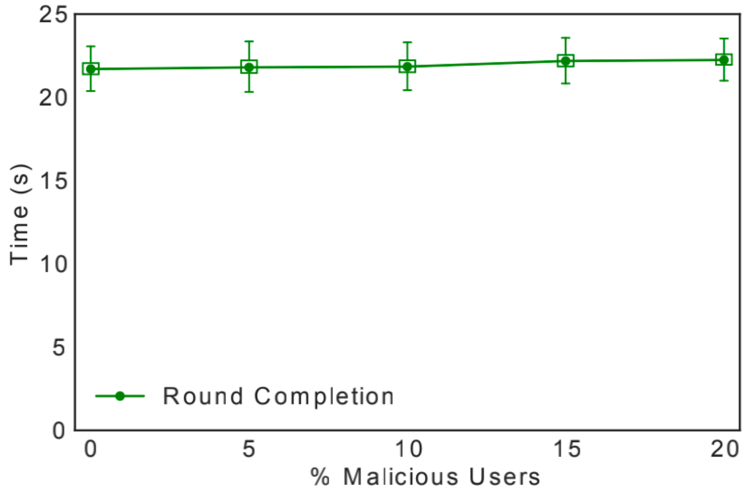



**Figure 8**: Latency for one round of Algorand with a varying fraction of malicious users, out of a total of 50,000 users.

# Conclusion

Algorand can scale well to millions of users.

Algorand produces no fork.

Algorand resilient to various types of attacks

**Thank you**

**Any Questions?**