Algorand: Scaling Byzantine Agreements for Cryptocurrencies

Yossi Gilad Rotem Hemo Silvio Micali Georgios Vlachos Nickolai Zeldovich

Abstract

Algorand is a new cryptocurrency system that can confirm transactions with latency on the order of a minute while scaling to many users [15]. Algorand ensures that users never have divergent views of confirmed transactions, even if some of the users are malicious and the network is partitioned. In contrast, existing cryptocurrencies allow for temporary forks and therefore require a long time, on the order of an hour, to confirm transactions with high confidence.

Algorand uses a new Byzantine Agreement (BA) protocol to reach consensus among users on the next set of transactions. To scale the consensus to many users, Algorand uses a novel mechanism based on Verifiable Random Functions that allows users to privately check whether they are selected to participate in the BA to agree on the next set of transactions, and to include a proof of their selection in their network messages. In Algorand's BA protocol, users do not keep any private state except for their private keys, which allows Algorand to replace participants immediately after they send a message. This mitigates targeted attacks on chosen participants after their identity is revealed.

We implement Algorand and evaluate its performance on 1,000 EC2 virtual machines, simulating up to 500,000 users. Experimental results show that Algorand confirms transactions in under a minute, achieves 30× Bitcoin's throughput, and incurs almost no penalty for scaling to more users.

1 Introduction

Cryptographic currencies such as Bitcoin can enable new applications, such as smart contracts [21, 41] and fair protocols [4], can simplify currency conversions [12], and can avoid trusted centralized authorities that regulate transactions. However, current proposals suffer from a trade-off between latency and confidence in a transaction. For example, achieving a high confidence that a transaction has been confirmed in Bitcoin requires about an hour long wait [7]. On the other hand, applications that require low latency cannot be certain that their transaction will be confirmed, and must trust the payer to not double-spend [37].

Double-spending is the core problem faced by any cryptocurrency, where an adversary holding $1 gives his $1 to two different users.[1] Cryptocurrencies prevent double-spending by reaching consensus on an ordered log ("block-chain") of transactions. Reaching consensus is difficult because of the open setting of cryptocurrencies. Since anyone should be able to participate, an adversary can create an arbitrary number of pseudonyms ("Sybils") [19] making it infeasible to rely on traditional consensus protocols [14] that require a fraction of honest users.

Bitcoin [33] and other cryptocurrencies [20, 44] address this problem using a proof-of-work (PoW) scheme, where users must repeatedly compute hashes to grow the block-chain, and the longest chain is considered authoritative. PoW ensures that an adversary does not gain any advantage by creating pseudonyms. However, PoW allows the possibility of *forks*, where two different block-chains have the same length, and neither one supersedes the other. Mitigating forks requires two unfortunate sacrifices: the time to grow the chain by one block must be reasonably high (10 minutes in Bitcoin), and applications must wait for several blocks in order to ensure their transaction remains on the authoritative chain (6 blocks are recommended in Bitcoin [7]). The result is that it takes about an hour to confirm a transaction in Bitcoin.

This paper presents Algorand, a new cryptocurrency system designed to confirm transactions on the order of one minute. The core of Algorand uses a Byzantine agreement protocol called $BA\star$ that scales to many users, which allows Algorand to reach consensus on a new block with low latency and without the possibility of forks. A key technique that makes $BA\star$ suitable for Algorand is the use of verifiable random functions (VRFs) [30] to perform a random selection of users in a private and non-interactive way, as we describe shortly. A synchronous version of $BA\star$ was presented at a workshop at a high level [3]; and a technical report [15] describes Algorand in detail.

Algorand faces three key challenges. First, Algorand must avoid Sybil attacks where an adversary creates many pseudonyms to influence the Byzantine agreement protocol. Second, $BA\star$ must scale to millions of users, which is far higher than the scale at which state-of-the-art Byzantine agreement protocols operate. Finally, Algorand must be resilient to denial-of-service (DoS) attacks: Algorand should continue to operate even if an adversary disconnects some of the users in the system.

Algorand addresses these challenges using several techniques, as follows.

Weighted users. To prevent Sybil attacks, Algorand assigns weights to each user. $BA\star$ is designed to guarantee consensus as long as a weighted fraction (a constant greater than 2/3) of the users are honest. In Algorand, we weigh users based on the money in their account. Thus, as long as more than some fraction (e.g., 2/3) of the money is owned by honest users, Algorand can avoid forks and double-spending.

Consensus by committee. $BA\star$ achieves scalability by choosing a committee—a small set of representatives ran-

[1] Cryptographic signatures on transactions can easily ensure that only an account owner can transfer money from that account.

domly selected from the total set of users—to run each step of its protocol. All other users observe the protocol messages, which allows them to learn the agreed-upon block. *BA*⋆ chooses committee members randomly among all users based on the users' weights. This allows Algorand to ensure that a sufficient fraction of committee members are honest. However, relying on a committee creates the possibility of targeted attacks against the chosen committee members.

Cryptographic sortition. To prevent an adversary from targeting committee members, *BA*⋆ selects committee members in a private and non-interactive way. This means that every user in the system can independently determine if they are chosen to be on the committee, by computing a function (a VRF [30]) of their private key and public information from the block-chain. If the function indicates that the user is chosen, it returns a short string that proves this user's committee membership to all other users, which the user can then include in network messages he sends. Since membership selection is non-interactive, an adversary does not know which user to target until that user starts participating in *BA*⋆.

Participant replacement. Finally, an adversary may target a committee member once that member sends a message in *BA*⋆. *BA*⋆ mitigates this attack by requiring committee members to speak just once. Thus, once a committee member sends his message (exposing his identity to an adversary), the committee member becomes irrelevant to *BA*⋆. *BA*⋆ achieves this property by avoiding any private state (except for the user's private key), which makes all users equally capable of participating, and by electing new committee members for each step of the Byzantine agreement protocol.

We implement a prototype of Algorand and *BA*⋆, and use it to empirically evaluate Algorand's performance. Experimental results running on 1,000 Amazon EC2 VMs demonstrate that Algorand can confirm a 1 MByte block of transactions in 40 seconds with 50,000 users, that Algorand's latency remains nearly constant when scaling to half a million users, that Algorand achieves 30× the transaction throughput of Bitcoin, and that Algorand achieves acceptable latency even in the presence of actively malicious users.

2 Related work

Proof-of-work. Bitcoin [33], the predominant cryptocurrency, uses proof-of-work to ensure that everyone agrees on the set of approved transactions; this approach is often called "Nakamoto consensus." Bitcoin must balance the length of time to compute a new block with the possibility of wasted work [33], and sets parameters to generate a new block every 10 minutes on average. Nonetheless, due to the possibility of forks, it is widely suggested that users wait for the block chain to grow by at least six blocks before considering their transaction to be confirmed [7]. This means transactions in Bitcoin take on the order of an hour to be confirmed. Many follow-on cryptocurrencies adopt Bitcoin's proof-of-work approach and inherit its limitations. The possibility of forks also makes it difficult for new users to bootstrap securely: an adversary that isolates the user's network can convince the user to use a particular fork of the blockchain [24].

By relying on Byzantine agreement, Algorand eliminates the possibility of forks, enables one to rely on a block as soon as it appears on the blockchain, and avoids the need to reason about mining strategies [8, 22, 38]. As a result, transactions are confirmed on the order of a minute.

Byzantine consensus. Byzantine agreement protocols have been used to replicate a service across a small group of servers, such as in PBFT [14]. Follow-on work has shown how to make Byzantine fault tolerance perform well and scale to dozens of servers [2, 16, 27]. One downside of Byzantine fault tolerance protocols used in this setting is that they require a fixed set of servers to be determined ahead of time; allowing anyone to join the set of servers would open up the protocols to Sybil attacks. These protocols also do not scale to the large number of users targeted by Algorand. *BA*⋆ is a Byzantine consensus protocol that does not rely on a fixed set of servers, which avoids the possibility of targeted attacks on well-known servers. By weighing users according to their currency balance, *BA*⋆ allows users to join the cryptocurrency without risking Sybil attacks, as long as the fraction of the money held by honest users is at least a constant greater than 2/3. *BA*⋆'s design also allows it to scale to many users (e.g., 500,000 shown in our evaluation) using VRFs to fairly select a random committee.

Most Byzantine consensus protocols require more than $2/3$ of servers to be honest, and Algorand's *BA*⋆ inherits this limitation (in the form of $2/3$ of the money being held by honest users). BFT2F [28] shows that it is possible to achieve "fork*-consensus" with just over half of the servers being honest, but fork*-consensus would allow an adversary to double-spend on the two forked block-chains, which Algorand avoids.

Honey Badger [31] demonstrated how Byzantine fault tolerance can be used to build a cryptocurrency. Specifically, Honey Badger designates a set of servers to be in charge of reaching consensus on the set of approved transactions. This allows Honey Badger to reach consensus within 2 minutes using 104 servers. One downside of this design is that the cryptocurrency is no longer decentralized; there are a fixed set of servers chosen when the system is first configured. Fixed servers are also problematic in terms of targeted attacks that either compromise the servers or disconnect them from the network. Algorand achieves a better performance (confirming transactions on the order of a minute) without having to choose a fixed set of servers ahead of time.

Pass and Shi's hybrid consensus [34] tries to refine the Honey Badger design to allow the set of servers to change over time. Pass and Shi suggest using Bitcoin's proof-of-work block chain to select a new set of servers every day, and then to use Byzantine consensus among those servers to confirm transactions for the duration of a day. Although this makes the set of Byzantine servers dynamic, it opens up the possibility

of forks, especially on network partitions, due to the use of proof-of-work consensus to agree on the set of servers; this problem cannot arise in Algorand.

Pass and Shi's paper acknowledges their design is secure only with respect to a "mildly adaptive" adversary that cannot compromise the selected servers within a day, and explicitly calls out the open problem of whether it is possible to handle fully adaptive adversaries. Ouroboros also relies on committees that are assumed to be incorruptible by the adversary in at least a day [25]. Algorand's $BA\star$ explicitly addresses this open problem by immediately replacing any chosen committee members. As a result, Algorand is not susceptible to either targeted compromises or targeted DoS attacks.

Stellar [29] takes an alternative approach to Byzantine consensus in a cryptocurrency, where each user can trust other users, forming a trust hierarchy. Consistency is ensured as long as all transactions share at least one transitively trusted user, and that user is honest. Algorand avoids this assumption, which means that users do not have to make complex trust decisions when configuring their client software.

Proof of stake. Algorand assigns weights to users proportionally to the monetary value they have in the system, inspired by proof-of-stake approaches [10]. There is a key difference, however, between Algorand using monetary value as weights and proof-of-stake cryptocurrencies. In a proof-of-stake cryptocurrency, a malicious leader (who assembles a new block) *can create a fork* in the network, but if caught (e.g., since two versions of the new block are signed with his key), the leader loses his money (this is the "stake"). The weights in Algorand, however, are only to ensure that the attacker cannot amplify his power by using pseudonyms; as long as the attacker controls less than 1/3 of the monetary value, Algorand can guarantee that the probability for forks is negligible. Algorand may be extended to "punish bad leaders", but this is not required to prevent forks.

Proof-of-stake avoids the computational overhead of proof-of-work and therefore allows reducing transaction confirmation time. However, realizing proof of stake in practice is challenging [5]. Since no work is involved in generating blocks, a malicious leader can announce one block, sell his credits in the currency (remove their stake), and then present some other block to isolated users. Therefore some proof-of-stake cryptocurrencies require a master key to periodically sign the correct branch of the ledger in order to mitigate forks [26]. This raises significant trust concerns regarding the currency, and had also caused accidental forks in the past [35]. Algorand answers this challenge by avoiding forks, even if the leader turns out to be malicious.

Trees and DAGs instead of chains. GHOST and SPECTRE are two recent proposals for increasing Bitcoin's throughput by replacing the underlying chain-structured ledger with a tree or directed acyclic graph (DAG) structures [39, 40], and resolving conflicts in the forks of these data structures. Both protocols rely on the Nakamoto consensus and are bound by the latency of its proof-of-work. By carefully designing the selection rule between two branches of the trees/DAGs, they are able to substantially increase the throughput, but not to eliminate the confirmation time users should wait before accepting transactions that appear on the ledger. In contrast, Algorand is focused on reducing transaction latency; in future work, it may be interesting to explore whether tree or DAG structures can similarly increase Algorand's throughput.

3 Goals and assumptions

Algorand achieves two important goals:

Safety. With overwhelming probability, $1 - 10^{-18}$, all users agree on the same transactions, and the network is always in a consistent state. More precisely, if a honest user accepts a transaction, then the same transaction is accepted by all honest users who see it. This holds even for isolated users that are disconnected from the network—e.g., by Eclipse attacks [24]. Accordingly, Algorand guarantees both low-latency and high-confidence transaction confirmation.

Liveness. In addition to safety, Algorand also makes progress (i.e., allows new transactions to be added to the log) under an additional assumptions about network reachability that we describe below. Algorand aims to reach consensus on a new set of transactions within roughly one minute.

Algorand makes standard cryptographic assumptions such as public-key signatures and hash functions. Algorand assumes that honest users run bug-free software. As mentioned earlier, Algorand assumes that the fraction of money held by honest users is above some threshold p (a constant greater than $2/3$), but that an adversary can participate in Algorand and own some money. We believe that this assumption is reasonable, since it means that in order to successfully attack Algorand, the attacker must invest substantial financial resources in it. Algorand assumes that an adversary can corrupt targeted users, but that an adversary cannot corrupt a large number of users that hold a significant fraction of the money (i.e., the amount of money held by honest, non-compromised users must remain over the threshold).

To ensure safety, Algorand makes no assumptions about the network; the attacker can have full control of the network. To ensure liveness, Algorand requires that more than a threshold of honest users can communicate with each other.

4 Overview

Algorand requires each user to have a public key.[2] Algorand maintains a log of transactions, called a blockchain. Each transaction is a payment signed by one user's public key transferring money to another user's public key. Algorand grows the blockchain in asynchronous *rounds*, similar to Bitcoin. In every round, a new block, containing a set of transactions, is appended to the blockchain.

[2] In the rest of this paper, we refer to Algorand software running on a user's computer as that user.

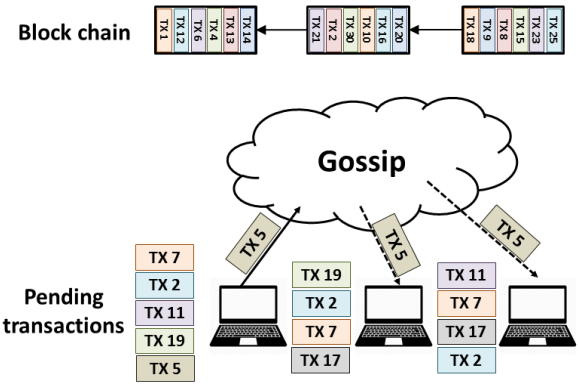

Figure 1: An overview of transaction flow in Algorand.

Algorand users communicate through a gossip protocol. The gossip protocol is used by users to submit new transactions. Each user collects a block of pending transactions that they hear about, in case they are chosen to propose the next block, as shown in Figure 1. Algorand uses *BA*⋆ to reach consensus on one of these pending blocks.

BA⋆ executes in *steps*, communicates over the same gossip protocol, and produces a new agreed-upon block. Once *BA*⋆ announces consensus, all transactions in this block are considered to be confirmed. The rest of this section provides a more detailed overview of Algorand's main components.

Gossip protocol. Algorand implements a gossip network (similar to Bitcoin and other P2P networks) where each user selects a small random set of peers to gossip messages to. To ensure messages cannot be forged, every message is signed by the private key of its original sender; other users check that the signature is valid before relaying it. To avoid loops, users do not relay the same message twice. Algorand implements gossip over TCP and weighs peer selection based on how much money they have, so as to mitigate pollution attacks.

Block proposal (§6). All Algorand users execute *cryptographic sortition* to determine whether they are selected to propose a block in a given round. We describe sortition in §5, but at a high level, sortition ensures that a small fraction of users are selected at random, weighed by their account balance, and provides each selected user with a *priority*, which can be compared between users, and a *proof* of the chosen user's priority. Selected users distribute their block of pending transactions through the gossip protocol, together with their priority and proof. To ensure that users converge on one block with high probability, blocks are prioritized based on the proposing user's priority, and users wait for a certain amount of time to receive the block.

Agreement using *BA*⋆ (§7). Block proposal does not guarantee that all users received the same block, and in fact, Algorand does not rely on the block proposal protocol for safety. To reach consensus on a single block, Algorand uses *BA*⋆. Each user initializes *BA*⋆ with the highest-priority block that they received. *BA*⋆ executes in repeated steps, illustrated in Figure 2. Each step begins with cryptographic sortition (§5), where all users check whether they have been selected as committee members in that step. Committee members then broadcast a message which includes their proof of selection. These steps repeat until, in some step of *BA*⋆, enough users in the committee reach consensus.[3] As discussed earlier, an important feature of *BA*⋆ is that committee members do not keep private state except their private keys, and so can be replaced after every step, to mitigate targeted attacks on them.

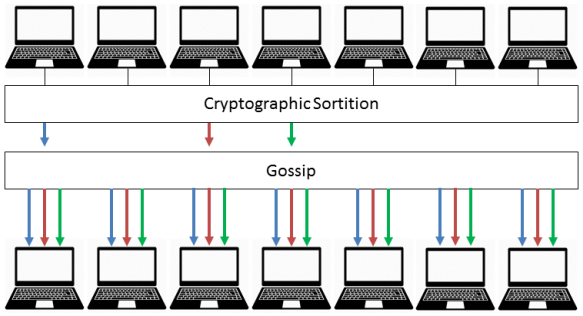

Figure 2: An overview of one step of *BA*⋆. To simplify the figure, each user is shown twice: once at the top of the diagram and once at the bottom. Each arrow color indicates a message from a particular user.

Efficiency. *BA*⋆ guarantees that if all honest users start with the same initial block (i.e., the user with the lowest block credential was honest), then *BA*⋆ establishes consensus over that block and terminates precisely in 3 steps. In the worst case of a particularly lucky adversary, all honest users reach consensus on the next block within expected 13 steps [15].

5 Cryptographic sortition

Cryptographic sortition is an algorithm for choosing a random subset of users according to per-user weights; that is, given a set of weights w_i and the weight of all users $W = \sum_j w_j$, the probability that user i is selected is proportional to w_i/W. The randomness in the sortition algorithm comes from a publicly-known random *seed*; we describe later how this seed is chosen. To allow a user to prove that they were chosen, sortition requires each user to have a public/private key pair, (pk_i, sk_i).

Sortition is implemented using verifiable random functions (VRFs) [30]. Informally, on any input string x, $VRF_{sk_i}(x)$ returns two values: a hash and a proof. The hash is an l-bit-long value that is uniquely determined by sk_i and x, but is indistinguishable from random to anyone that does not know sk_i. The proof π enables anyone that knows pk_i to check that the hash indeed corresponds to x, without having to know sk_i.

Using VRFs, Algorand implements cryptographic sortition as shown in Algorithm 1. Sortition requires a *role* parameter that distinguishes the different roles that a user may be selected for; for example, the user may be selected to propose a block in some round, or they may be selected to be the

[3] Steps are not synchronized across users; each user checks for selection as soon as he observes the previous step had ended.

member of the committee at a certain step of $BA\star$. Algorand assigns a threshold τ_{role} for each role to set the expected number of users selected for that role.

procedure Sortition($sk_i, seed, role, w_i, W$):
$\langle hash, \pi \rangle \leftarrow \text{VRF}_{sk_i}(seed||role)$
if $hash < 2^l \cdot \tau_{role} \cdot \frac{w_i}{W}$ **then return** $\langle hash, \pi \rangle$;
else return $\perp$;

Algorithm 1: The cryptographic sortition algorithm. l is the length (in bits) of the VRF output.

Given a random seed, the VRF outputs a pseudo-random hash value, which is essentially uniformly distributed between 0 and $2^l - 1$. By checking if the resulting hash is below a threshold based on τ_{role} and w_i, the algorithm ensures that, on average, τ_{role} users are selected, according to their weights (w_i for user i). If a user is not selected, sortition returns $\perp$. However, an adversary that does not know sk_i cannot guess if user i is chosen (more precisely, cannot guess any better than just by randomly guessing based on the weights). The pseudocode for verifying the sortition proof, shown in Algorithm 2 follows the same structure.

procedure VerifySort($pk_i, hash, \pi, seed, role, w_i, W$):
if $hash >= 2^l \cdot \tau_{role} \cdot \frac{w_i}{W}$ **then return** *false* ;
else return $VerifyVRFproof_{pk_i}(hash, \pi, seed||role)$;

Algorithm 2: Pseudocode for verifying sortition of user i.

5.1 Choosing the seed

Sortition requires a seed that is chosen at random and publicly known. In the context of Algorand, each round requires a new seed that is publicly known by everyone for that round, but cannot be controlled by the adversary; otherwise, an adversary may be able to choose a seed that favors the selection of corrupted users.

The seed for Algorand's round r is determined using VRFs with the seed of the previous round $r-1$. More specifically, during the block proposal stage of round $r-1$, every user u selected for block proposal also computes a proposed seed for round r as $\langle seed_r, \pi \rangle \leftarrow \text{VRF}_{sk_u}(seed_{r-1}||r)$. Algorand requires that sk_u be chosen by u well in advance of $seed_{r-1}$ being determined (§5.2). This ensures that even if u is malicious, the resulting $seed_r$ is pseudo-random.

This seed (and the corresponding VRF proof π) is included in every proposed block, so that once Algorand reaches agreement on the block for round $r-1$, everyone knows $seed_r$ at the start of round r. If the block does not contain a valid seed (e.g., because the block was proposed by a malicious user and π is not a valid VRF proof), users treat the entire proposed block as if it were empty, and use a cryptographic hash function H (which we assume is a random oracle) to compute the associated seed for round r as $seed_r = H(seed_{r-1}||r)$. The value of $seed_0$, which bootstraps seed selection, can be chosen at random at the start of Algorand by the initial participants using distributed random number generation [13].

5.2 Choosing sk_u well in advance of the seed

Computing $seed_r$ requires that every user's secret key sk_u is chosen well in advance of $seed_{r-1}$. To enforce this, whenever Algorand performs cryptographic sortition in round r, it uses the keys (and the associated weights) from round $r-k$. This ensures that, as long as k is suitably large, an adversary u choosing a key sk_u in round $r-k$ cannot predict the seed for round r. This is the case because the seed for round r depends on the secret keys of the agreed-upon block proposers for rounds $r-k$ through r, and it is highly unlikely that all of those k block proposers will be controlled by the attacker.

Our technical report [15: §6.4] formally analyzes the seed selection process, and proves that $k = \lceil \log_{\frac{1}{2}} F \rceil$, where F is a negligible probability, deemed acceptable for forks. For Algorand's targeted probability (10^{-18}), $k = 60$ is sufficient to avoid an adversary influencing the seed.

6 Block proposal

To ensure that some block is proposed in each round, Algorand sets the sortition threshold for the block-proposal role, $\tau_{proposer}$, to be greater than 1 (although Algorand will reach consensus on at most one of these proposed blocks). Our technical report proves that choosing $\tau_{proposer} = 26$ ensures that a reasonable number of proposers (at least one, and no more than 70, as a plausible upper bound) are chosen with very high probability ($1 - 10^{-12}$) [15].

Minimizing unnecessary block transmissions. One risk of choosing several block proposers is that each will gossip their own proposed block. For a large block (say, 1 MByte), this can incur a significant communication cost. To reduce this cost, the sortition hash is used to prioritize block proposals (smaller hash means higher priority). Algorand users discard messages about blocks that do not have the highest priority seen by that user so far. Algorand also gossips two kinds of messages: one contains just the priorities and proofs of the chosen block proposers (from cryptographic sortition), and the other contains the entire block, which also includes the proposer's sortition hash, and proof. The first kind of message is small (about 200 Bytes), and propagates quickly through the gossip network. These messages enable most users to learn who is the highest priority proposer, and thus discard blocks of lower-priority proposers.

Waiting for block proposals. Each user must wait a certain amount of time to receive block proposals via the gossip protocol. Choosing this time interval does not impact Algorand's safety guarantees but is important for performance. Waiting a short amount of time will mean no received proposals. If the user receives no block proposals, he or she initializes $BA\star$ with the empty block, and if many users do so, Algorand will reach consensus on an empty block. On the other

hand, waiting too long will receive all block proposals but also unnecessarily increase the confirmation latency.

To determine the appropriate amount of time to wait for block proposals, we consider the plausible scenarios that a user might find themselves in. When a user starts waiting for block proposals for round r, they may be one of the first users to reach consensus in round $r-1$. Since that user was able to complete round $r-1$, sufficiently many users have sent a message for the last step of *BA*$\star$ in that round, and therefore, most of the network is at most one step behind this user. Thus, the user must somehow wait for others to finish the last step of *BA*$\star$ from round $r-1$. At this point, some proposer in round r that happens to have the highest priority will gossip their priority and proof message, and the user must somehow wait to receive that message. Then, the user can simply wait until they receive the block corresponding to the highest priority proof (with a timeout λ_{BLOCK}, on the order of a minute, after which the user will fall back to the empty block).

It is impossible for a user to wait exactly the correct amount for the first two steps of the above scenario. Thus, Algorand estimates these quantities (λ_{STEPVAR}, the variance in how long it takes different users to finish the last step of *BA*$\star$, and $\lambda_{\text{PRIORITY}}$, the time taken to gossip the priority and proof message), and waits for $\lambda_{\text{STEPVAR}} + \lambda_{\text{PRIORITY}}$ time to identify the highest priority. §10 experimentally shows that these parameters are, conservatively, 10 seconds and 5 seconds, respectively. As mentioned above, Algorand would remain safe even if these estimates were inaccurate.

Malicious proposers. Even if some block proposers are malicious, the worst case scenario is that they trick different Algorand users into initializing *BA*$\star$ with different blocks. This could in turn cause Algorand to reach consensus on an empty block, and possibly take additional steps in doing so. However, it turns out that even this scenario is relatively unlikely. In particular, if the adversary is *not* the highest priority proposer in a round, then the highest priority proposer will gossip a consistent version of their block to all users. If the adversary is the highest priority proposer in a round, they can propose the empty block, and thus prevent any real transactions from being confirmed. However, this happens with probability of at most $1-p$, by Algorand's assumption that at least $p > 2/3$ of the weighted user are honest.

7 *BA*$\star$

The execution of *BA*$\star$ consists of two phases. In the first phase, *BA*$\star$ reduces the problem of agreeing on a block to agreement on a binary value. In the second phase, *BA*$\star$ reaches agreement on a bit that indicates whether the consensus is on a block from the first phase, or whether it is on $\perp$ (the empty block). Each phase consists of several interactive *steps*; the first phase always takes two steps, and the second phase takes one step if the highest-priority block proposer was honest (sent the same block to all users), and an expected 11 steps in the worst case of a malicious highest-priority proposer colluding with a large fraction of committee participants at every step. To decide whether to move from one step to the next, *BA*$\star$ counts the "votes" committee members cast and checks if they exceed a threshold fraction T of the expected number of selected committee members, denoted by $\tau_{committee}$; we discuss how to choose T and $\tau_{committee}$ in §7.1.

BA$\star$ provides the following guarantees:

1. All honest users agree on the same block. This ensures Algorand's safety.
2. If more than a fraction T of the weighted users are both honest and start with the same block, then all honest users agree on *that block*. This ensures Algorand's progress.

A key aspect of *BA*$\star$'s design is that it keeps no secrets, except for user private keys. This allows any user observing the messages to "passively participate" in the protocol, i.e., verify signatures, count votes, and reach the agreement decision.

Initialization (Alg. 3). *BA*$\star$ starts running on each user's computer for a given round with Initialize(), shown in Algorithm 3. The *block* argument to Initialize() is the highest-priority block received during block proposal. As long as the block is valid (has the seed for the next round, and does not contain double-spending transactions), *BA*$\star$ tries to reach consensus on the hash of the block (which makes the reduction phase more efficient than running directly on the block). If *BA*$\star$ reaches consensus on a hash of a non-empty block, then at least a T fraction of the weighted users initialized *BA*$\star$ with that block, and that block is widely available even if a few users did not receive it during block proposal. The initialize procedure also sets a timeout for terminating the first step of *BA*$\star$, this is λ_{BLOCK} (the time it takes to propagate a block, since some users may receive the block and start the first step at most λ_{BLOCK} after this user) plus λ_{STEP}, the time Algorand allocates to run any other step in *BA*$\star$.

```
procedure Initialize(round, block):
state.r ← round
state.step ← REDUCTION_STEP_ONE
state.done ← false
// terminate first step with TIMEOUT after λ_BLOCK + λ_STEP
SetTimer(StepCompletion(TIMEOUT), λ_BLOCK + λ_STEP)
if DoubleSpends(block) or ¬ValidSeed(block) then
    state.block ← ⊥
    StartStep(⊥)
else
    state.block ← block
    StartStep(H(state.block))
end
```

Algorithm 3: Initializing *BA*$\star$ for round *round* with the proposed *block*. H is a cryptographic hash function.

Starting a step (Alg. 4). Algorithm 4 shows the pseudocode for StartStep() which starts every step of *BA*⋆. This code invokes Sortition() from Algorithm 1 to check if the user is chosen to participate in the committee at each step. If the user is chosen for this step, the user gossips a signed message containing the value passed to StartStep(); depending on the phase, this can either be the hash of the block (for the reduction phase) or either ACCEPT or REJECT (for the binary consensus phase).

procedure StartStep(v):
// check if already handled enough messages to complete
$v' \leftarrow \arg\max_{value}$ *state.counts*[*state.r*][*state.step*][*value*]
if *state.counts*[*state.r*][*state.step*][v'] $> T \cdot \tau_{committee}$ **then**
  return StepCompletion(v') ;
// check if user is in committee using Sortition (Alg. 1)
role $\leftarrow$ *"committee"*||*state.step*
ret $\leftarrow$ Sortition(*state.sk*, $seed_{state.r-1}$, *role*, *state.weight*[*state.pk*], *state.W*)
// only committee members originate a message
if *ret* $\neq \perp$ **then**
  $\langle sorthash, \pi \rangle \leftarrow ret$
  Gossip($\langle$*state.pk*, Signed$_{state.sk}$(*state.r*, *state.step*, *sorthash*, π, v)$\rangle$)
end

Algorithm 4: Starting a step of *BA*⋆, voting for value v. *state.sk* and *state.pk* is the user's private/public key pair.

Counting votes (Alg. 5). In each step, MessageHandler() counts how many votes (with valid signatures and proofs) each user received for each value, as shown in Algorithm 5. As soon as one value has more than $T \cdot \tau_{committee}$ votes, MessageHandler() calls StepCompletion(). ($\tau_{committee}$ is the expected number of users that Sortition() selects for the committee, and is the same for each step). Note that all the processing in Algorithm 5 can be performed by all users as they relay messages in the gossip network. This allows us to replace committee members at each step.

Step completion (Alg. 6). Algorithm 6 shows StepCompletion(), which is called when either a user receives $T \cdot \tau_{committee}$ votes for the same value (in which case v is this value), or there was a time-out (in which case v is TIMEOUT). StepCompletion distinguishes between the two phases of *BA*⋆. If another step is required (concensus was not yet reached), then the completion procedure sets a timer for that new step (allowing it to run up to λ_{STEP} seconds), and the initializes that step.

The first two steps implement the Turpin and Coan reduction [42] from an arbitrary value agreement (the hash of the next block), to a binary value agreement. In the first step of the reduction the committee members gossip the hash of the block they received (when Initialize calls StartStep). In the second step they gossip a hash that received at least

procedure MessageHandler(m):
// ignore from past rounds; defer future rounds
if *m.r* $<$ *state.r* **then return**;
if *m.r* $>$ *state.r* **then return** HANDLELATER;
// discard messages with invalid signature or proof
if *VerifySignature*(m) $\neq$ OK **then return**;
if *VerifySort*(*m.src*, *m.sorthash*, *m.*π, $seed_{m.r-1}$, *"committee"*||*m.step*, *state.weight*[*m.src*], *state.W*) $\neq$ OK **then return**;
// discard duplicate senders; track all messages
if *m.src* $\in$ *state.senders*[*m.r*][*m.step*] **then return**;
state.senders[*m.r*][*m.step*].add(*m.src*)
state.messages[*m.r*][*m.step*].add(m)
// count *src*'s vote
state.counts[*m.r*][*m.step*][*m.value*]++
// relay the message to others in the P2P network
Gossip(m)
// if this step is complete, call StepCompletion
if *state.counts*[*m.r*][*m.step*][*m.value*] $> T \cdot \tau_{committee}$ **then**
  if *state.step* = *m.step* **then**
    StepCompletion(*m.value*) ;
end

Algorithm 5: MessageHandler(), called for every incoming message m.

$T \cdot \tau_{committee}$ of the votes in the first step, or $\perp$ if no such hash exists. When REDUCTION_STEP_TWO completes, *BA*⋆ starts agreement on a bit, which is the core of the *BA*⋆ protocol and described next. Importantly, the Turpin-Coan reduction does not modify any private state, the property that *BA*⋆ needs to replace committee members.

In all following steps committee members vote to ACCEPT the hash value, or REJECT and use the empty block. The ConcludeBinaryStep function, explained next, is called to check whether agreement on either ACCEPT or REJECT was reached, and if not, to decide on the value for the next vote.

Agreement on a bit (Alg. 7). To agree on a bit, StepCompletion calls ConcludeBinaryStep, shown in Algorithm 7, which receives the value for which the user counted at least $T \cdot \tau_{committee}$ votes in the current step. If neither ACCEPT nor REJECT received that many votes, then the input for ConcludeBinaryStep is TIMEOUT. ConcludeBinaryStep sets *state.done* to mark whether agreement was established. It also returns an ACCEPT or REJECT value, which should be the user's vote in the next step (if selected to be in the committee).

To agree on a bit, *BA*⋆ repeats three steps until reaching consensus: BIN_STEP_ACCEPT, BIN_STEP_REJECT, and BIN_STEP_COIN. At the end of each step, a user who has seen more than $T \cdot \tau_{committee}$ votes for either ACCEPT or REJECT adopts that value for the next step. Importantly, we must select the committee to be large enough such that it is extremely unlikely that some users adopt ACCEPT while

procedure StepCompletion(v):

if *state.step* = REDUCTION_STEP_ONE **then**
  // no state changes, just re-gossip the value v
  // when starting the next step (at the end of this func.)
else if *state.step* = REDUCTION_STEP_TWO **then**
  // save the hash; we next agree whether to accept it
  state.hash_of_block $\leftarrow v$
  if *state.hash_of_block* $= \perp$ **then** $v \leftarrow$ REJECT;
  else $v \leftarrow$ ACCEPT;
  // proceed to start binary agreement
else // completed a binary step
  // if finished, ConcludeBinaryStep sets *state.done*
  $v \leftarrow$ ConcludeBinaryStep(v)
end
ClearTimer()
if *state.done* **then**
  // reached consensus
  if v = ACCEPT **then**
    NotifyConsensus(state.hash_of_block) ;
  else NotifyConsensus($\perp$) ;
else // we need another step, consensus not yet reached
  // terminate new step if not complete in λ_{STEP} sec
  SetTimer(StepCompletion(TIMEOUT), λ_{STEP})
  state.step++
  StartStep(v)
end

Algorithm 6: Step completion. Called from MessageHandler with v, the "popular value" that received more than $T \cdot \tau_{committee}$ votes, or TIMEOUT.

procedure ConcludeBinaryStep(v):

if *state.step* = BIN_STEP_ACCEPT *(mod 3)* **then**
  if v = REJECT **then return** REJECT;
  else if v = ACCEPT **then** *state.done* $\leftarrow$ *true* ;
  return ACCEPT
else if *state.step* = BIN_STEP_REJECT *(mod 3)* **then**
  if v = ACCEPT **then return** ACCEPT;
  else if v = REJECT **then** *state.done* $\leftarrow$ *true* ;
  return REJECT
else // BIN_STEP_COIN *(mod 3)*
  if $v \in \{\text{ACCEPT}, \text{REJECT}\}$ **then return** v;
  else return CommonCoin();
end

Algorithm 7: Completing a step in the binary agreement phase. The input is $v \in \{\text{ACCEPT}, \text{REJECT}, \text{TIMEOUT}\}$.

others adopt REJECT. We discuss this requirement, and set the appropriate committee size, in §7.1.

BIN_STEP_ACCEPT guarantees that if all honest users entered with ACCEPT, then all of them learn that agreement was reached on the hash of the block. BIN_STEP_REJECT guarantees that if all honest users entered the step with REJECT, then they all agree on REJECT.

In BIN_STEP_COIN all users compute a random "common coin," meaning a binary value (ACCEPT or REJECT) that is predominantly the same for all users. Although this may sound circular, the users need not reach formal consensus on this common coin; its purpose is to, informally, push a "stuck consensus" towards either ACCEPT or REJECT.

Any honest user who did not observe more than $T \cdot \tau_{committee}$ votes for either ACCEPT or REJECT adopts the value of the common coin. If only some of the honest users counted more than $T \cdot \tau_{committee}$ votes for ACCEPT (respectively REJECT), and the common coin is ACCEPT (respectively REJECT), then all honest users will be in agreement on ACCEPT (respectively REJECT). In this case, *BA*$\star$ will terminate in the following BIN_STEP_ACCEPT (respectively BIN_STEP_REJECT).[4] Thus, if all users were to use the same randomly-chosen coin value, they would reach consensus with probability 1/2. By repeating these steps, the probability of consensus quickly approaches 1.

We next show how to implement a coin that, with probability $p > \frac{2}{3}$, is both random and is observed by most users. This leads to consensus with probability $\frac{1}{2} \cdot p > \frac{1}{3}$ at each BIN_STEP_COIN.

Implementing the common coin (Alg. 8). Every BIN_STEP_COIN, *BA*$\star$ attempts to synchronize users on a coin that is decided at random and is publicly available to everyone. To implement this coin we take advantage of the VRF-based committee member hashes attached to all of the messages. Every user sets the common coin to be the least-significant bit of the lowest hash it observed in this step, as shown in Algorithm 8. Notice that hashes are random (since they are produced by a VRF), so their least-significant bits are also random. The common coin is used only when BIN_STEP_COIN times out, giving sufficient time for all votes to propagate through the network. If the committee member with the lowest sortition hash is honest, then all users that received his message observe the same coin.

procedure CommonCoin():
if *state.messages*[*state.r*][*state.step*].*empty*() **then**
  return REJECT
end
$m \leftarrow$ *MinSorthash*(*state.messages*[*state.r*][*state.step*])
if *m.sorthash = 0 mod 2* **then return** ACCEPT;
else return REJECT;

Algorithm 8: Computing a coin common to all users.

If a malicious committee-member happens to hold the lowest sortition hash, then he might send it to only some users. This may result in users observing different coin values, and thus will not help in reaching consensus. However, since sortition hashes are pseudo-random, the probability that an honest user has the lowest hash is p, and thus there is at least

[4] As we explained above, the committee is sufficiently large such that the case where some users observed enough votes to adopt ACCEPT while others observed enough votes to adopt REJECT is extremely unlikely.

a $p > \frac{2}{3}$ probability that the lowest sortition hash holder will be honest.

7.1 Analysis

The fraction $p > \frac{2}{3}$ of weighted honest users in Algorand must translate into a "sufficiently honest" committee for *BA*⋆. *BA*⋆ has two parameters at its disposal: $\tau_{committee}$, which controls the *expected* committee size, and T, which controls the number of votes needed to reach consensus ($T \cdot \tau_{committee}$). We would like T to be as small as possible for liveness, but we also need T to be sufficiently large to ensure safety. Moreover, the smaller T is, the larger $\tau_{committee}$ needs to be, to ensure that an adversary does not obtain enough votes by chance.

To make these constraints precise, let us denote the number of honest committee members by g and the malicious ones by b; in expectation, $b+g = \tau_{committee}$, but $b+g$ can vary since it is chosen by sortition. To ensure safety, *BA*⋆ requires $\frac{1}{2}g + b \leq T \cdot \tau_{committee}$. This requirement guarantees that two different values cannot both be "popular enough" to trigger the call to StepCompletion() in MessageHandler(). To ensure that Algorand can reach agreement on the next block and make progress, we must ensure that $g > T \cdot \tau_{committee}$.

Due to the probabilistic nature of how committee members are chosen, there is always some small chance that the b and g for some step fail to satisfy the above constraints, and *BA*⋆'s goal is to make this probability negligible. We choose T such that the liveness constraint is satisfied with probability $1 - 10^{-12}$, assuming all users are online; that is, *BA*⋆ may fail to make progress because some users are disconnected, but *BA*⋆ should not fail to make progress because of an unfortunate sortition outcome. Figure 3 plots the $\tau_{committee}$ that is needed to satisfy the safety constraint, as a function of p, for two different probabilities of safety violation, 10^{-12} and 10^{-18}. The results show that, as p approaches $\frac{2}{3}$, the committee size grows quickly. However, at $p = 80\%$, $\tau_{committee} = 4{,}000$ can ensure safety with probability of $1 - 10^{-18}$ (using T=0.69).

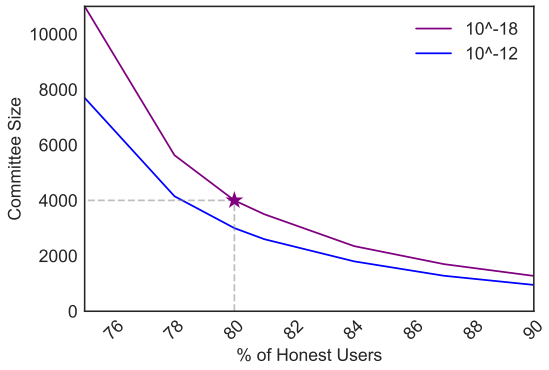

Figure 3: The committee size, $\tau_{committee}$, sufficient to limit the probability of violating safety to either 10^{-12} or 10^{-18}. The x-axis specifies p, the fraction of honest users. ⋆ marks the parameters selected in our implementation.

Given a suitable choice of $\tau_{committee}$, *BA*⋆ ensures safety even if an adversary has full control of the network, can create network partitions, and can propose different blocks in each partition. As a result, Algorand guarantees that its blockchain will not fork even in the presence of such a powerful attacker. In Algorand, the attacker cannot spoof credentials of committee members (from one partition) so as to convince users (in another partition) that consensus exists over the block they received. If a partition does not have sufficiently many users to approve the next block, then users running *BA*⋆ in that partition will continuously time out *BA*⋆ steps (calling StepCompletion with TIMEOUT) and neither approve nor reject new blocks. Once sufficiently many users can communicate to establish consensus over the next block, those users will reach agreement and continue to approve blocks in the following rounds. Detailed proofs of *BA*⋆'s correctness, including safety and liveness, are provided in the technical report [15].

8 Algorand

Building Algorand on top of the primitives we have described so far requires Algorand to address a number of higher-level issues, which this section discusses.

Bootstrapping the system. In order to deploy Algorand, a common genesis block must be provided to all users, along with the initial cryptographic sortition seed. As §5 mentions, at round r, Algorand must use user's public keys (and weights) from round $r-k$ to ensure the seeds remain pseudo-random. We resolve this issue at bootstrapping by repeating an empty block k times after the genesis block.

Bootstrapping new users. Users that join the system need to learn the current state of the system, which is defined to be the result of a chain of *BA*⋆ consensus outcomes. To help users catch up, Algorand generates a *certificate* for every block that was agreed upon by *BA*⋆ (including empty blocks). The certificate is an aggregate of the messages from *BA*⋆ that is sufficient to allow any user to reach the same conclusion by processing these messages. Importantly, the users must check the sortition hashes and proofs just like in Algorithm 5, and that all messages in the certificate are for the same Algorand round and *BA*⋆ step. If the certificate accepts the new block, then this step is BIN_STEP_ACCEPT (mod 3) , or if the certificate indicates REJECT, then this step is BIN_STEP_REJECT (mod 3).

To support certificates, binary ACCEPT messages include the hash of the block that is being decided (*state.hash_of_block* in Algorithm 6). ACCEPT certificates (those containing $\lfloor T \cdot \tau_{committee} \rfloor + 1$ ACCEPT votes) are provided with the block that was accepted, so that users can validate that its hash appears in all signed ACCEPT messages in the certificate. REJECT certificates tell the user that the block in consensus for that round was the empty block.

Certificates allow new users to validate prior blocks. Users validate blocks in order, starting from the genesis block. This ensures that the user knows the correct weights for verifying sortition proofs in any given round.

One potential risk created by the use of certificates is that an adversary can provide a certificate that appears to show that $BA\star$ completed after some large number of steps. This gives the adversary a chance to find a $BA\star$ step number in which the adversary controls more than a threshold of the selected committee members (and to then create a signed certificate using their private keys). We set the committee size to be sufficiently large to ensure the attacker has negligible probability of finding such a step number. For $\tau_{committee} > 1,000$, the probability of this attack is less than 2^{-166} at every step, making this attack infeasible.

Forward security. Attackers may attempt to corrupt users over time. More precisely, since identities of committee members are revealed after they send a message, an attacker can observe which users took part in the committee at one execution step in $BA\star$, and attempt to compromise their machine later so as to obtain their private keys. If an attacker manages to obtain more than a T fraction of the keys for the relatively small committee, he could fake a certificate and create a fork.

Algorand resists such attacks by having users commit to a different message-signing public key for every step, and publish their commitments on the blockchain. We use identity-based encryption (IBE) [11] to allow users to concisely commit to a long sequence of public keys (IBE user names of the form $\langle r,s\rangle$ where r and s are the round and step number), and to allow users to forget private keys immediately after they use them. This ensures that, by the time a user's identity is exposed, they have already deleted the corresponding private key. To limit the number of private keys users must create, Algorand must make another assumption: that there are no network partitions that preclude most Algorand users from communicating for a very long epoch (e.g., a month). We believe that this assumption is reasonable (e.g., the Internet has not had such a large-scale outage in its entire history, even though there have been instances of individual users or even countries being cut off for weeks). The technical report [15: §5.2] describes this construction in more detail.

Gossiping blocks. Algorand's block proposal protocol (§6) assumed that chosen users can gossip new blocks before an adversary can learn the user's identity and mount a targeted DoS attack against them. In practice, Algorand's blocks are larger than the maximum packet size, so it is inevitable that some packets from a chosen block proposer will be sent before others. A particularly fast adversary could take advantage of this to immediately DoS any user that starts sending multiple packets, on the presumption that the user is a block proposer.

Formally, this means that Algorand's liveness guarantees are slightly different in practice: instead of providing liveness in the face of immediate targeted DoS attacks, Algorand ensures liveness as long as an adversary cannot mount a targeted DoS attack within the time it takes for the victim to send a block over a TCP connection (a few seconds). We believe this does not matter significantly; an adversary with such a quick reaction time likely also has broad control over the network, and thus can prevent Algorand nodes from communicating at all. Another approach may be to rely on Tor [18] to make it difficult for an adversary to quickly disconnect a user.

"Rich users." To ensure that users with more than $\tau_{committee}/\mathit{all\ money}$ fraction of the money in Algorand are not under-represented in the committee, these (few) users split their money to several keys, each holding less of this fraction of money. Hence, such users may have more than one vote in the committee.

Scalability. The communication costs for each Algorand user depend on the expected size of the committee and the number of block proposers, which Algorand sets through $\tau_{proposer}$ and $\tau_{committee}$ (independent of the number of users). As more users join, it takes a message longer to disseminate in the gossip network. Algorand forms a random network graph (each user connects to random peers), and our theoretical analysis suggests that dissemination time grows with the diameter of the graph [15], which is logarithmic in the number of users [36]. Experiments confirm that Algorand's performance is only slightly affected by more users (§10).

9 Implementation

We implemented a prototype of Algorand in C++, consisting of approximately 5400 lines of code. We use Boost ASIO library for networking. Signatures and VRFs are implemented over Curve 25519 [6], and we use SHA-256 for a hash function. We use the VRF construction outlined in Goldberg et al [23: §4]. Our prototype does not implement forward security, but we expect it would have a negligible performance effect.

In our implementation each user connects to 4 random peers, accepts incoming connections from other peers, and gossips messages to all of them. This gives us 8 peers on average. We currently provide each user with an "address book" file listing the IP address and port number for every user's public key. In a real-world deployment we imagine users could gossip this information, signed by their keys, or distribute it via a public bulletin board.

Figure 4 shows the parameters in our prototype of Algorand; we experimentally validate the timeout parameters in §10. $p = 80\%$ means that an adversary would need to control 20% of Algorand's currency in order to create a fork. By analogy, in the US, the top 0.1% of people own about 20% of the wealth [32], so the richest 300,000 people would have to collude to create a fork.

$\lambda_{\text{PRIORITY}}$ should be large enough to allow block proposers to gossip their priorities and proofs. Measurements of message propagation in Bitcoin's network [17] suggest that gossiping 1 KB to 90% of the Bitcoin peer-to-peer network takes about 1 second. We conservatively set $\lambda_{\text{PRIORITY}}$ to 5 seconds.

λ_{BLOCK} ensures that Algorand can make progress even if the block proposer does not send the block. Our experiments

Parameter	Meaning	Value
p	assumed fraction of honest weighted users	80%
k	round look-back for sortition	60 (§5.1)
$\tau_{proposer}$	expected # of block proposers	26 (§6)
$\tau_{committee}$	expected # of committee members	4,000 (§7.1)
T	threshold of $\tau_{committee}$ for $BA\star$	69% (§7.1)
$\lambda_{\text{PRIORITY}}$	time to gossip sortition proofs	5 seconds
λ_{BLOCK}	timeout for receiving a block	1 minute
λ_{STEP}	timeout for $BA\star$ step	20 seconds
λ_{STEPVAR}	estimate of $BA\star$ completion time variance	10 seconds

Figure 4: Implementation parameters

(§10) show that about 10 seconds suffices to gossip a 1 MB block. We conservatively set λ_{BLOCK} to be a minute.

λ_{STEP} should be high enough to allow users to receive messages from committee members, but low enough to allow Algorand to make progress (move to the next step) if it does not hear from sufficiently many committee members. We conservatively set λ_{STEP} to 20 seconds. We set λ_{STEPVAR}, the estimated variance in $BA\star$ completion times, to 10 seconds.

10 Evaluation

Our evaluation quantitatively answers the following:

§10.1 What is the latency that Algorand can achieve for confirming transactions, and how does it scale as the number of users grows?

§10.2 What throughput can Algorand achieve in terms of transactions per second?

§10.3 What are Algorand's CPU, bandwidth, and storage costs?

§10.4 How does Algorand perform when users go offline or misbehave?

§10.5 Does Algorand choose reasonable timeout parameters?

To answer these questions, we deploy our prototype of Algorand on Amazon's EC2 using 1,000 m4.xlarge virtual machines (VMs),[5] each of which has 4 cores and up to 1 Gbps network throughput. To measure the performance of Algorand with a large number of users, we run multiple Algorand users (each user is a process) on the same VM. By default, we run 50 users per VM, and users propose a 1 MByte block. To simulate commodity network links, we cap the bandwidth for each Algorand process to 20 Mbps. To model network latency we use inter-city latency and jitter measurements [43] and assign each machine to one of 20 major cities around the world; latency within the same city is modeled as negligible. We assign an equal share of money to each user; the distribution of money does not change the number of messages that each user processes. Graphs in the rest of this section plot the time it takes for Algorand to complete an entire round, and include the minimum, median, maximum, 25th, and 75th percentile times across all users.

[5] We planned to use 2,000 VMs to simulate 1,000,000 users, but Amazon EC2 did not have sufficiently many instances available.

10.1 Latency

Figure 5 shows results with the number of users varying from 5,000 to 50,000 (by varying the number of active VMs from 100 to 1,000). The results show that Algorand can confirm transactions in well under a minute, and the latency is near-constant as the number of users grows.

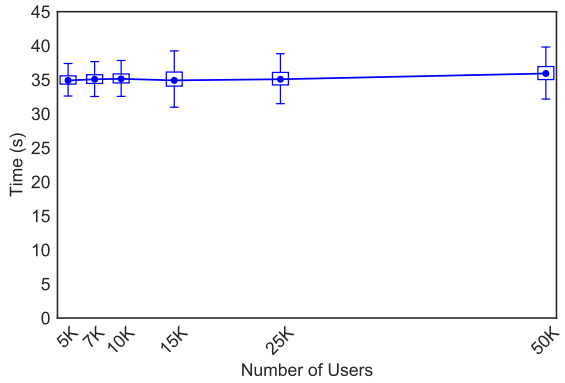

Figure 5: Latency for one round of Algorand, with 5,000 to 50,000 users.

To determine if Algorand continues to scale to even more users, we run an experiment with 500 Algorand user processes per VM. This configuration runs into two bottlenecks: CPU time and bandwidth. Most of the CPU time is spent verifying signatures and VRFs. To alleviate this bottleneck in our experimental setup, for this experiment we replace verifications with sleeps of the same duration. We are unable to alleviate the bandwidth bottleneck, since each VM's network interface is maxed out; instead, we increase λ_{STEP} to 1 minute.

Figure 6 shows the results of this experiment, scaling the number of users from 5,000 to 500,000 (by varying the number of VMs from 10 to 1,000). The latency in this experiment is about 3× higher than in Figure 5, even for the same number of users, owing to the bandwidth bottleneck. However, the scaling performance remains flat all the way to 500,000 users, suggesting that Algorand scales well.

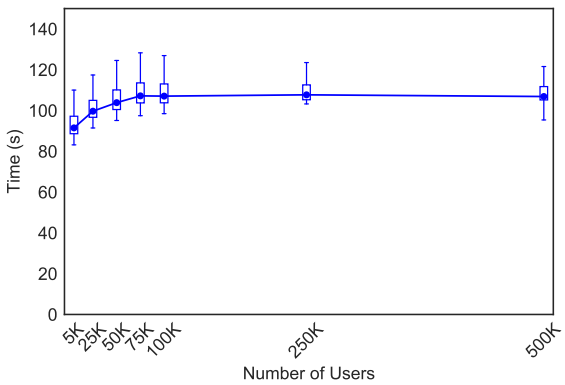

Figure 6: Latency for one round of Algorand in a configuration with 500 users per VM, using 10 to 1,000 VMs.

10.2 Throughput

In the following set of experiments we deploy 50,000 users on our 1,000 VMs (50 users per machine). Figure 7 shows the results with a varying block size. The figure breaks the Algorand round into two parts: block proposal (§6, at the bottom of the graph) is the time it takes a user to obtain the proposed block, and agreement (§7, at the top of the graph) is the time it takes for *BA*⋆ to complete. The block proposal time for small block sizes is dominated by the $\lambda_{\text{PRIORITY}} + \lambda_{\text{STEPVAR}}$ wait time. For large block sizes, the time to gossip the large block contents dominates.

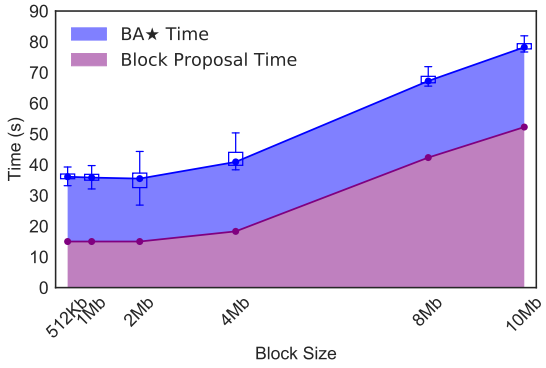

Figure 7: Latency for one round of Algorand as a function of the block size.

The results show that, at its lowest latency, Algorand commits a 2 MByte block in about 40 seconds, which means it can commit 180 MBytes of transactions per hour. For comparison, Bitcoin commits a 1 MByte block every 10 minutes, which means it can commit 6 MBytes of transactions per hour [9]. As Algorand's block size grows past 2 MBytes, Algorand achieves higher throughput at the cost of some increase to latency. For example, with a 10 MByte block size, Algorand commits about 450 MBytes of transactions per hour.

10.3 Costs of running Algorand

Users running Algorand incur CPU, network, and storage costs. The CPU cost of running Algorand is modest; when running 50 users per VM, CPU usage on the 4-core VM was about 70%, meaning each Algorand process uses about 5% of a core. In terms of bandwidth, each user in our experiment with 1 MByte blocks and 50,000 users use about 6 Mbit/sec of bandwidth. Algorand also stores block certificates in order to prove to new users that a block is committed. This storage cost is in addition to the blocks themselves. Each block certificate is 600 KBytes, independent of the block size; for 1 MByte blocks, this would be a ∼60% storage overhead.

10.4 Misbehaving and offline users

Figure 8 shows the results for different fractions of offline users (with 50,000 total users). The results show that Algorand performs well as long as enough users are present to reach consensus. Otherwise, *BA*⋆ hangs until enough users come online; our experiment confirms this when we run with 35% of the users being offline. Algorand's performance improves slightly as users go offline, because fewer messages are relayed by the gossip protocol.

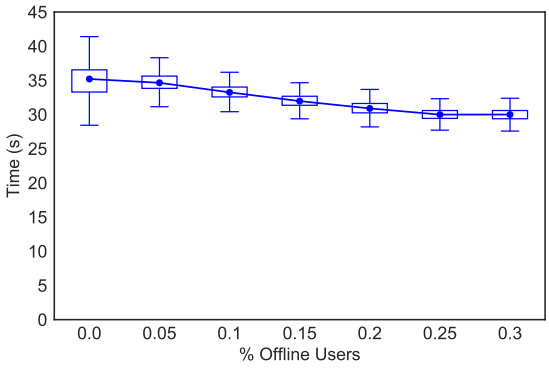

Figure 8: Latency for one round of Algorand with a varying fraction of offline users, and 50,000 total users.

Algorand's safety is guaranteed by *BA*⋆ (§7), but proving this experimentally would require testing all possible attacker strategies, which is infeasible. However, to experimentally show that our Algorand prototype handles malicious users, we choose one particular attack strategy. We force the block proposer with the highest priority to equivocate about the proposed block: namely, the proposer sends one version of the block to half of its peers, and another version to others. Malicious users that are chosen to be part of the *BA*⋆ committee vote for both blocks. Figure 9 shows how Algorand's performance is affected by the weighted fraction of malicious users. The results show that, at least empirically for this particular attack, Algorand is not significantly affected.

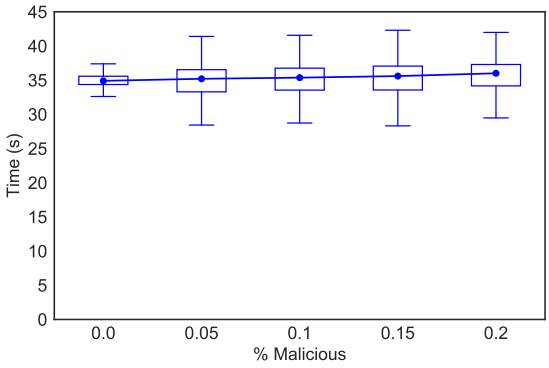

Figure 9: Latency for one round of Algorand with a varying fraction of malicious users, out of a total of 50,000 users.

10.5 Timeout parameters

The graphs shown in this section confirm that *BA*⋆ steps finish in well under λ_{STEP} (20 seconds), that the difference between 25th and 75th percentiles of *BA*⋆ completion times is under λ_{STEPVAR} (10 seconds), and that blocks are gossiped within

λ_{BLOCK} (1 minute). We separately measure the time taken to propagate a block proposer's priority and proof; it is consistently around 1 second, well under $\lambda_{\text{PRIORITY}}$ (5 seconds), confirming the measurements by Decker and Wattenhofer [17].

10.6 Ongoing work on improving performance

We continue to work on improving our implementation of Algorand, and expect to reduce latency and increase throughput.

11 Conclusion

Algorand is a new cryptocurrency that confirms transactions on the order of a minute with a negligible probability of forking. Algorand's design is based on a cryptographic sortition mechanism combined with the *BA*$\star$ Byzantine agreement protocol. Algorand avoids targeted attacks at chosen participants using participant replacement at every step. Experimental results with a prototype of Algorand demonstrate that it achieves sub-minute latency and 30$\times$ the throughput of Bitcoin, and scales well to 500,000 users.

References

[1] *Proceedings of the 21st Annual International Cryptology Conference (CRYPTO)*, Santa Barbara, CA, Aug. 2001.

[2] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 59–74, Brighton, UK, Oct. 2005.

[3] Anonymous. Anonymized talk describing *BA*$\star$, Jan. 2017.

[4] I. Bentov and R. Kumaresan. How to use Bitcoin to design fair protocols. In *Proceedings of the 34th Annual International Cryptology Conference (CRYPTO)*, Santa Barbara, CA, Aug. 2014.

[5] I. Bentov, A. Gabizon, and A. Mizrahi. Cryptocurrencies without proof of work. In *Proceedings of the 2016 Financial Cryptography and Data Security Conference*, 2016.

[6] D. J. Bernstein. Curve25519: New diffie-hellman speed records. In *PKC*, pages 207–228, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-33852-9. doi: 10.1007/11745853_14. URL http://dx.doi.org/10.1007/11745853_14.

[7] Bitcoin Wiki. Confirmation. https://en.bitcoin.it/wiki/Confirmation, 2017.

[8] BitcoinWiki. Mining hardware comparison, 2016. https://en.bitcoin.it/wiki/Mining_hardware_comparison.

[9] BitcoinWiki. Bitcoin scalability. https://en.bitcoin.it/wiki/Scalability, 2017.

[10] BitcoinWiki. Proof of stake. https://en.bitcoin.it/wiki/Proof_of_Stake, 2017.

[11] D. Boneh and M. K. Franklin. Identity-based encryption from the Weil pairing. In *Proceedings of the 21st Annual International Cryptology Conference (CRYPTO)* CRY [1].

[12] G. Brockman. Stellar, July 2014. https://stripe.com/blog/stellar.

[13] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols. In *Proceedings of the 21st Annual International Cryptology Conference (CRYPTO)* CRY [1], pages 524–541.

[14] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, pages 173–186, New Orleans, LA, Feb. 1999.

[15] J. Chen and S. Micali. Algorand. *CoRR*, abs/1607.01341, 2017. URL http://arxiv.org/abs/1607.01341.

[16] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 153–168, Boston, MA, Apr. 2009.

[17] C. Decker and R. Wattenhofer. Information propagation in the Bitcoin network. In *Proceedings of the 13th IEEE International Conference on Peer-to-Peer Computing*, Sept. 2013.

[18] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th Usenix Security Symposium*, pages 303–320, San Diego, CA, Aug. 2004.

[19] J. R. Douceur. The Sybil attack. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, MA, Mar. 2002.

[20] Ethereum Foundation. Ethereum, 2016. https://www.ethereum.org/.

[21] Ethereum Foundation. Create a democracy contract in Ethereum, 2016. https://www.ethereum.org/dao.

[22] I. Eyal and E. G. Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *Proceedings of the 2013 Financial Cryptography and Data Security Conference*, Mar. 2014.

[23] S. Goldberg, M. Naor, D. Papadopoulos, and L. Reyzin. NSEC5 from elliptic curves: Provably preventing DNSSEC zone enumeration with shorter responses. Cryptology ePrint Archive, Report 2016/083, Mar. 2016. http://eprint.iacr.org/.

[24] E. Heilman, A. Kendler, A. Zohar, and S. Goldberg. Eclipse attacks on Bitcoin's peer-to-peer network. In *Proceedings of the 24th Usenix Security Symposium*, pages 129–144, Washington, DC, Aug. 2015.

[25] A. Kiayias, I. Konstantinou, A. Russell, B. David, and R. Oliynykov. A provably secure proof-of-stake blockchain protocol. Cryptology ePrint Archive, Report 2016/889, 2016. http://eprint.iacr.org/.

[26] S. King and S. Nadal. PPCoin: Peer-to-peer crypto-currency with proof-of-stake, Aug. 2012. https://peercoin.net/assets/paper/peercoin-paper.pdf.

[27] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. L. Wong. Zyzzyva: Speculative Byzantine fault tolerance. *ACM Transactions on Computer Systems*, 27(4):7:1–39, 2009.

[28] J. Li and D. Mazières. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)*, Cambridge, MA, Apr. 2007.

[29] D. Mazières. The Stellar consensus protocol: A federated model for internet-level consensus. https://www.stellar.org/papers/stellar-consensus-protocol.pdf, 2014.

[30] S. Micali, M. O. Rabin, and S. P. Vadhan. Verifiable random functions. In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, New York, NY, Oct. 1999.

[31] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. The Honey Badger of BFT protocols. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, pages 31–42, Vienna, Austria, Oct. 2016.

[32] A. Monaghan. US wealth inequality: top 0.1% worth as much as the bottom 90%, Nov. 2014. https://www.theguardian.com/business/2014/nov/13/us-wealth-inequality-top-01-worth-as-much-as-the-bottom-90.

[33] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008.

[34] R. Pass and E. Shi. Hybrid consensus: Efficient consensus in the permissionless model. Cryptology ePrint Archive, Report 2016/917, 2016. http://eprint.iacr.org/.

[35] Peercointalk. Peercoin invalid checkpoint. https://www.peercointalk.org/t/invalid-checkpoint/3691, 2015.

[36] O. Riordan and N. Wormald. The diameter of sparse random graphs. *Combinatorics, Probability and Computing*, 19(5-6):835–926, Nov. 2010.

[37] P. Rizzo. BitGo launches "instant" Bitcoin transaction tool, Jan. 2016. http://www.coindesk.com/bitgo-instant-bitcoin-transaction-tool/.

[38] J. Rubin. The problem of ASICBOOST, Apr. 2017. http://www.mit.edu/~jlrubin/public/pdfs/Asicboost.pdf.

[39] Y. Sompolinsky and A. Zohar. Secure high-rate transaction processing in Bitcoin. In *Proceedings of the 2015 Financial Cryptography and Data Security Conference*, 2015.

[40] Y. Sompolinsky, Y. Lewenberg, and A. Zohar. SPECTRE: A fast and scalable cryptocurrency protocol. Cryptology ePrint Archive, Report 2016/1159, 2016. http://eprint.iacr.org/.

[41] N. Szabo. Smart contracts: Formalizing and securing relationships on public networks. *First Monday*, 2(9), Sept. 1997. http://firstmonday.org/ojs/index.php/fm/article/view/548/469.

[42] R. Turpin and B. A. Coan. Extending binary Byzantine agreement to multivalued Byzantine agreement. *Information Processing Letters*, 18(2):73–76, Feb. 1984.

[43] WonderNetwork. Global ping statistics: Ping times between WonderNetwork servers, Apr. 2017. https://wondernetwork.com/pings.

[44] Zerocoin Electric Coin Company. ZCash: All coins are created equal, 2017. https://z.cash.