ALGORAND*

Jing Chen
Computer Science Department
Stony Brook University
Stony Brook, NY 11794, USA
jingchen@cs.stonybrook.edu

Silvio Micali
CSAIL
MIT
Cambridge, MA 02139, USA
silvio@csail.mit.edu

Abstract

A public ledger is a tamperproof sequence of data that can be read and augmented by everyone. Public ledgers have innumerable and compelling uses. They can secure, in plain sight, all kinds of transactions —such as titles, sales, and payments— in the exact order in which they occur. Public ledgers not only curb corruption, but also enable very sophisticated applications —such as cryptocurrencies and smart contracts. They stand to revolutionize the way a democratic society operates. As currently implemented, however, they scale poorly and cannot achieve their potential.

Algorand is a truly democratic and efficient way to implement a public ledger. Unlike prior implementations based on proof of work, it requires a negligible amount of computation, and generates a transaction history that will not "fork" with overwhelmingly high probability.

Algorand is based on (a novel and super fast) message-passing Byzantine agreement.

For concreteness, we shall describe Algorand only as a money platform.

1 Introduction

Money is becoming increasingly virtual. It has been estimated that about 80% of United States dollars today only exist as ledger entries [5]. Other financial instruments are following suit.

In an ideal world, in which we could count on a universally trusted central entity, immune to all possible cyber attacks, money and other financial transactions could be solely electronic. Unfortunately, we do not live in such a world. Accordingly, decentralized cryptocurrencies, such as Bitcoin [29], and "smart contract" systems, such as Ethereum, have been proposed [4]. At the heart of these systems is a shared *ledger* that reliably records a sequence of transactions,

*This is the more formal (and asynchronous) version of the ArXiv paper by the second author [24], a paper itself based on that of Gorbunov and Micali [18]. Algorand's technologies are the object of the following patent applications: US62/117,138 US62/120,916 US62/142,318 US62/218,817 US62/314,601 PCT/US2016/018300 US62/326,865 62/331,654 US62/333,340 US62/343,369 US62/344,667 US62/346,775 US62/351,011 US62/653,482 US62/352,195 US62/363,970 US62/369,447 US62/378,753 US62/383,299 US62/394,091 US62/400,361 US62/403,403 US62/410,721 US62/416,959 US62/422,883 US62/455,444 US62/458,746 US62/459,652 US62/460,928 US62/465,931

as varied as payments and contracts, in a tamperproof way. The technology of choice to guarantee such tamperproofness is the *blockchain*. Blockchains are behind applications such as cryptocurrencies [29], financial applications [4], and the Internet of Things [3]. Several techniques to manage blockchain-based ledgers have been proposed: *proof of work* [29], *proof of stake* [2], *practical Byzantine fault-tolerance* [8], or some combination.

Currently, however, ledgers can be inefficient to manage. For example, Bitcoin's *proof-of-work* approach (based on the original concept of [14]) requires a vast amount of computation, is wasteful and scales poorly [1]. In addition, it *de facto* concentrates power in very few hands.

We therefore wish to put forward a new method to implement a public ledger that offers the convenience and efficiency of a centralized system run by a trusted and inviolable authority, without the inefficiencies and weaknesses of current decentralized implementations. We call our approach *Algorand*, because we use algorithmic randomness to select, based on the ledger constructed so far, a set of *verifiers* who are in charge of constructing the next block of valid transactions. Naturally, we ensure that such selections are provably immune from manipulations and unpredictable until the last minute, but also that they ultimately are universally clear.

Algorand's approach is quite democratic, in the sense that neither in principle nor *de facto* it creates different classes of users (as "miners" and "ordinary users" in Bitcoin). In Algorand "all power resides with the set of all users".

One notable property of Algorand is that its transaction history may fork only with very small probability (e.g., one in a trillion, that is, or even 10^{-18}). Algorand can also address some legal and political concerns.

The Algorand approach applies to blockchains and, more generally, to any method of generating a tamperproof sequence of blocks. We actually put forward a new method —alternative to, and more efficient than, blockchains— that may be of independent interest.

1.1 Bitcoin's Assumption and Technical Problems

Bitcoin is a very ingenious system and has inspired a great amount of subsequent research. Yet, it is also problematic. Let us summarize its underlying assumption and technical problems —which are actually shared by essentially all cryptocurrencies that, like Bitcoin, are based on *proof-of-work*.

For this summary, it suffices to recall that, in Bitcoin, a user may own multiple public keys of a digital signature scheme, that money is associated with public keys, and that a payment is a digital signature that transfers some amount of money from one public key to another. Essentially, Bitcoin organizes all processed payments in a chain of blocks, $B_1, B_2, \ldots$, each consisting of multiple payments, such that, all payments of B_1, taken in any order, followed by those of B_2, in any order, etc., constitute a sequence of valid payments. Each block is generated, on average, every 10 minutes.

This sequence of blocks is a *chain*, because it is structured so as to ensure that any change, even in a single block, percolates into all subsequent blocks, making it easier to spot any alteration of the payment history. (As we shall see, this is achieved by including in each block a *cryptographic hash* of the previous one.) Such block structure is referred to as a *blockchain*.

Assumption: Honest Majority of Computational Power Bitcoin assumes that no malicious entity (nor a coalition of coordinated malicious entities) controls the majority of the computational power devoted to block generation. Such an entity, in fact, would be able to modify the blockchain,

and thus re-write the payment history, as it pleases. In particular, it could make a payment $\wp$, obtain the benefits paid for, and then "erase" any trace of $\wp$.

Technical Problem 1: Computational Waste Bitcoin's proof-of-work approach to block generation requires an extraordinary amount of computation. Currently, with just a few hundred thousands public keys in the system, the top 500 most powerful supercomputers can only muster a mere 12.8% percent of the total computational power required from the Bitcoin players. This amount of computation would greatly increase, should significantly more users join the system.

Technical Problem 2: Concentration of Power Today, due to the exorbitant amount of computation required, a user, trying to generate a new block using an ordinary desktop (let alone a cell phone), expects to lose money. Indeed, for computing a new block with an ordinary computer, the expected cost of the necessary electricity to power the computation exceeds the expected reward. Only using *pools* of specially built computers (that do nothing other than "mine new blocks"), one might expect to make a profit by generating new blocks. Accordingly, today there are, *de facto*, two disjoint classes of users: ordinary users, who only make payments, and specialized mining pools, that only search for new blocks.

It should therefore not be a surprise that, as of recently, the total computing power for block generation lies within just five pools. In such conditions, the assumption that a majority of the computational power is honest becomes less credible.

Technical Problem 3: Ambiguity In Bitcoin, the blockchain is not necessarily unique. Indeed its latest portion often *forks*: the blockchain may be —say— $B_1, \ldots, B_k, B'_{k+1}, B'_{k+2}$, according to one user, and $B_1, \ldots, B_k, B''_{k+1}, B''_{k+2}, B''_{k+3}$ according another user. Only after several blocks have been added to the chain, can one be reasonably sure that the first $k+3$ blocks will be the same for all users. Thus, one cannot rely right away on the payments contained in the last block of the chain. It is more prudent to wait and see whether the block becomes sufficiently deep in the blockchain and thus sufficiently stable.

Separately, *law-enforcement* and *monetary-policy* concerns have also been raised about Bitcoin.[1]

1.2 Algorand, in a Nutshell

Setting Algorand works in a very tough setting. Briefly,

(a) *Permissionless and Permissioned Environments.* Algorand works efficiently and securely even in a totally permissionless environment, where arbitrarily many users are allowed to join the system at any time, without any vetting or permission of any kind. Of course, Algorand works even better in a *permissioned* environment.

[1]The (pseudo) anonymity offered by Bitcoin payments may be misused for money laundering and/or the financing of criminal individuals or terrorist organizations. Traditional banknotes or gold bars, that in principle offer perfect anonymity, should pose the same challenge, but the physicality of these currencies substantially slows down money transfers, so as to permit some degree of monitoring by law-enforcement agencies.

The ability to "print money" is one of the very basic powers of a nation state. In principle, therefore, the massive adoption of an independently floating currency may curtail this power. Currently, however, Bitcoin is far from being a threat to governmental monetary policies, and, due to its scalability problems, may never be.

(b) *Very Adversarial Environments.* Algorand withstands a very powerful Adversary, who can

(1) *instantaneously* corrupt *any user* he wants, at *any time* he wants, provided that, in a permissionless environment, 2/3 of the money in the system belongs to honest user. (In a permissioned environment, irrespective of money, it suffices that 2/3 of the users are honest.)

(2) *totally control and perfectly coordinate* all corrupted users; and

(3) *schedule the delivery of all messages,* provided that each message m sent by a honest user reaches 95% of the honest users within a time λ_m, which solely depends on the size of m.

Main Properties Despite the presence of our powerful adversary, in Algorand

- *The amount of computation required is minimal.* Essentially, no matter how many users are present in the system, each of fifteen hundred users must perform at most a few seconds of computation.

- *A New Block is Generated in less than 10 minutes, and will* de facto *never leave the blockchain.* For instance, in expectation, the time to generate a block in the first embodiment is less than $\Lambda + 12.4\lambda$, where Λ is the time necessary to propagate a block, in a peer-to-peer gossip fashion, *no matter what block size one may choose*, and λ is the time to propagate 1,500 200B-long messages. (Since in a truly decentralized system, Λ essentially is an intrinsic latency, in Algorand the limiting factor in block generation is network speed.) *The second embodiment has actually been tested experimentally (by ?), indicating that a block is generated in less than 40 seconds.*

  In addition, Algorand's blockchain *may fork only with negligible probability* (i.e., less than one in a trillion), and thus users can relay on the payments contained in a new block as soon as the block appears.

- *All power resides with the users themselves.* Algorand is a truy distributed system. In particular, there are no exogenous entities (as the "miners" in Bitcoin), who can control which transactions are recognized.

Algorand's Techniques.

1. A New and Fast Byzantine Agreement Protocol. Algorand generates a new block via a new cryptographic, *message-passing*, binary Byzantine agreement (BA) protocol, $BA^\star$. Protocol $BA^\star$ not only satisfies some additional properties (that we shall soon discuss), but is also very fast. Roughly said, its binary-input version consists of a 3-step loop, in which a player i sends a *single* message m_i to all other players. Executed in a complete and synchronous network, with more than 2/3 of the players being honest, with probability $> 1/3$, after each loop the protocol ends in agreement. (We stress that protocol $BA^\star$ satisfies the original definition of Byzantine agreement of Pease, Shostak, and Lamport [31], without any weakenings.)

Algorand leverages this binary BA protocol to reach agreement, in our different communication model, on each new block. The agreed upon block is then *certified*, via a prescribed number of digital signature of the proper verifiers, and propagated through the network.

2. Cryptographic Sortition. Although very fast, protocol $BA^\star$ would benefit from further speed when played by millions of users. Accordingly, Algorand chooses the players of $BA^\star$ to be

a much smaller subset of the set of all users. To avoid a different kind of concentration-of-power problem, each new block B^r will be constructed and agreed upon, via a new execution of $BA^\star$, by a separate set of *selected verifiers*, SV^r. In principle, selecting such a set might be as hard as selecting B^r directly. We traverse this potential problem by an approach that we term, embracing the insightful suggestion of Maurice Herlihy, *cryptographic sortition*. Sortition is the practice of selecting officials at random from a large set of eligible individuals [6]. (Sortition was practiced across centuries: for instance, by the republics of Athens, Florence, and Venice. In modern judicial systems, random selection is often used to choose juries. Random sampling has also been recently advocated for elections by David Chaum [9].) In a decentralized system, of course, choosing the random coins necessary to randomly select the members of each verifier set SV^r is problematic. We thus resort to cryptography in order to select each verifier set, from the population of all users, in a way that is guaranteed to be automatic (i.e., requiring no message exchange) and random. In essence, we use a cryptographic function to automatically determine, from the previous block B^{r-1}, a user, the *leader*, in charge of proposing the new block B^r, and the verifier set SV^r, in charge to reach agreement on the block proposed by the leader. Since malicious users can affect the composition of B^{r-1} (e.g., by choosing some of its payments), we specially construct and use additional inputs so as to prove that the leader for the rth block and the verifier set SV^r are indeed randomly chosen.

3. The Quantity (Seed) Q^r. We use the the last block B^{r-1} in the blockchain in order to automatically determine the next verifier set and leader in charge of constructing the new block B^r. The challenge with this approach is that, by just choosing a slightly different payment in the previous round, our powerful Adversary gains a tremendous control over the next leader. Even if he only controlled only 1/1000 of the players/money in the system, he could ensure that all leaders are malicious. (See the Intuition Section 4.1.) This challenge is central to all proof-of-stake approaches, and, to the best of our knowledge, it has not, up to now, been satisfactorily solved.

To meet this challenge, we purposely construct, and continually update, a separate and carefully defined quantity, Q^r, which *provably* is, not only unpredictable, but also not influentiable, by our powerful Adversary. We may refer to Q^r as the rth *seed*, as it is from Q^r that Algorand selects, via secret cryptographic sortition, all the users that will play a special role in the generation of the rth block.

4. Secret Crytographic Sortition and Secret Credentials. Randomly and unambiguously using the current last block, B^{r-1}, in order to choose the verifier set and the leader in charge of constructing the new block, B^r, is not enough. Since B^{r-1} must be known before generating B^r, the last non-influentiable quantity Q^{r-1} contained in B^{r-1} must be known too. Accordingly, so are the verifiers and the leader in charge to compute the block B^r. Thus, our powerful Adversary might immediately corrupt *all of them*, before they engage in any discussion about B^r, so as to get full control over the block they certify.

To prevent this problem, leaders (and actually verifiers too) *secretly* learn of their role, but can compute a proper *credential*, capable of proving to everyone that indeed have that role. When a user privately realizes that he is the leader for the next block, first he secretly assembles his own proposed new block, and then disseminates it (so that can be certified) together with his own credential. This way, though the Adversary will immediately realize who the leader of the next block is, and although he can corrupt him right away, it will be too late for the Adversary to influence the choice of a new block. Indeed, he cannot "call back" the leader's message no more

than a powerful government can put back into the bottle a message virally spread by WikiLeaks.

As we shall see, we cannot guarantee leader uniqueness, nor that everyone is sure who the leader is, including the leader himself! But, in Algorand, unambiguous progress will be guaranteed.

5. Player Replaceability. After he proposes a new block, the leader might as well "die" (or be corrupted by the Adversary), because his job is done. But, for the verifiers in SV^r, things are less simple. Indeed, being in charge of certifying the new block B^r with sufficiently many signatures, they must first run Byzantine agreement on the block proposed by the leader. The problem is that, no matter how efficient it is, $BA^\star$ requires *multiple* steps and the honesty of $> 2/3$ of its players. This is a problem, because, for efficiency reasons, the player set of $BA^\star$ consists the small set SV^r randomly selected among the set of all users. Thus, our powerful Adversary, although unable to corrupt 1/3 of *all the users*, can certainly corrupt *all members of* SV^r!

Fortunately we'll prove that protocol $BA^\star$, executed by propagating messages in a peer-to-peer fashion, is *player-replaceable*. This novel requirement means that the protocol correctly and efficiently reaches consensus even if each of its step is executed by a totally new, and randomly and independently selected, set of players. Thus, with millions of users, each small set of players associated to a step of $BA^\star$ most probably has empty intersection with the next set.

In addition, the sets of players of different steps of $BA^\star$ will probably have totally different *cardinalities*. Furthermore, the members of each set do not know who the next set of players will be, and do not secretly pass any internal state.

The replaceable-player property is actually crucial to defeat the dynamic and very powerful Adversary we envisage. We believe that replaceable-player protocols will prove crucial in lots of contexts and applications. In particular, they will be crucial to execute securely small sub-protocols embedded in a larger universe of players with a dynamic adversary, who, being able to corrupt even a small fraction of the total players, has no difficulty in corrupting all the players in the smaller sub-protocol.

An Additional Property/Technique: Lazy Honesty A honest user follows his prescribed instructions, which include being online and run the protocol. Since, Algorand has only modest computation and communication requirement, being online and running the protocol "in the background" is not a major sacrifice. Of course, a few "absences" among honest players, as those due to sudden loss of connectivity or the need of rebooting, are automatically tolerated (because we can always consider such few players to be temporarily malicious). Let us point out, however, that Algorand can be simply adapted so as to work in a new model, in which honest users to be offline most of the time. Our new model can be informally introduced as follows.

> *Lazy Honesty.* Roughly speaking, a user i is lazy-but-honest if (1) he follows all his prescribed instructions, when he is asked to participate to the protocol, and (2) he is asked to participate to the protocol only rarely, and with a suitable advance notice.

With such a relaxed notion of honesty, we may be even more confident that honest people will be at hand when we need them, and Algorand guarantee that, when this is the case,

The system operates securely even if, at a given point in time,
the majority of the participating players are malicious.

1.3 Closely Related work

Proof-of-work approaches (like the cited [29] and [4]) are quite orthogonal to our ours. So are the approaches based on message-passing Byzantine agreement or practical Byzantine fault tolerance (like the cited [8]). Indeed, these protocols cannot be run among the set of all users and cannot, in our model, be restricted to a suitably small set of users. In fact, our powerful adversary my immediately corrupt all the users involved in a small set charged to actually running a BA protocol.

Our approach could be considered related to proof of stake [2], in the sense that users' "power" in block building is proportional to *the money they own in the system* (as opposed to —say— to the money they have put in "escrow").

The paper closest to ours is the Sleepy Consensus Model of Pass and Shi [30]. To avoid the heavy computation required in the proof-of-work approach, their paper relies upon (and kindly credits) Algorand's secret cryptographic sortition. With this crucial aspect in common, several significant differences exist between our papers. In particular,

(1) *Their setting is only permissioned.* By contrast, Algorand is also a permissionless system.

(2) *They use a Nakamoto-style protocol, and thus their blockchain forks frequently.* Although dispensing with proof-of-work, in their protocol a secretly selected leader is asked to elongate the longest valid (in a richer sense) blockchain. Thus, forks are unavoidable and one has to wait that the block is sufficiently "deep" in the chain. Indeed, to achieve their goals with an adversary capable of adaptive corruptions, they require a block to be $poly(N)$ deep, where N represents the total number of users in the system. Notice that, even assuming that a block could be produced in a minute, if there were $N = 1M$ users, then one would have to wait for about 2M years for a block to become N^2-deep, and for about 2 years for a block to become N-deep. By contrast, Algorand's blockchain forks only with negligible probability, even though the Adversary corrupt users immediately and adaptively, and its new blocks can immediately be relied upon.

(3) *They do not handle individual Byzantine agreements.* In a sense, they only guarantee "eventual consensus on a growing sequence of values". Theirs is a *state replication* protocol, rather than a BA one, and cannot be used to reach Byzantine agreement on an individual value of interest. By contrast, Algorand can also be used only once, if so wanted, to enable millions of users to quickly reach Byzantine agreement on a specific value of interest.

(4) *They require weakly synchronized clocks.* That is, all users' clocks are offset by a small time δ. By contrast, in Algorand, clocks need only have (essentially) the same "speed".

(5) *Their protocol works with lazy-but-honest users or with honest majority of online users.* They kindly credit Algorand for raising the issue of honest users going offline en masse, and for putting forward the lazy honesty model in response. Their protocol not only works in the lazy honesty model, but also in their *adversarial sleepy model*, where an adversary chooses which users are online and which are offline, provided that, at all times, the majority of online users are honest.[2]

[2]The original version of their paper actually considered only security in their adversarial sleepy model. The original version of Algorand, which precedes theirs, also explicitly envisaged assuming that a given majority of the online players is always honest, but explicitly excluded it from consideration, in favor of the lazy honesty model. (For instance, if at some point in time half of the honest users choose to go off-line, then the majority of the users on-line may very well be malicious. Thus, to prevent this from happening, the Adversary should *force* most of his corrupted players to go off-line too, which clearly is against his own interest.) Notice that a protocol with a majority of lazy-but-honest players works just fine if the majority of the users on-line are always malicious. This is so, because a sufficient number of honest players, knowing that they are going to be crucial at some rare point in time, will elect not to go off-line in those moments, nor can they be forced off-line by the Adversary, since he does not know who the crucial honest players might be.

(6) *They require a simple honest majority.* By contrast, the current version of Algorand requires a 2/3 honest majority.

Another paper close to us is *Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol*, by Kiayias, Russell, David, and Oliynykov [20]. Also their system appeared after ours. It also uses crytpographic sortition to dispense with proof of work in a provable manner. However, their system is, again, a Nakamoto-style protocol, in which forks are both unavoidable and frequent. (However, in their model, blocks need not as deep as the sleepy-consensus model.) Moreover, their system relies on the following assumptions: in the words of the authors themselves, "(1) the network is highly synchronous, (2) the majority of the selected stakeholders is available as needed to participate in each epoch, (3) the stakeholders do not remain offline for long periods of time, (4) the adaptivity of corruptions is subject to a small delay that is measured in rounds linear in the security parameter." By contrast, Algorand is, with overwhelming probability, fork-free, and does not rely on any of these 4 assumptions. In particular, in Algorand, the Adversary is able to instantaneously corrupt the users he wants to control.

2 Preliminaries

2.1 Cryptographic Primitives

Ideal Hashing. We shall rely on an efficiently computable cryptographic hash function, H, that maps arbitrarily long strings to binary strings of fixed length. Following a long tradition, we model H as a *random oracle*, essentially a function mapping each possible string s to a randomly and independently selected (and then fixed) binary string, $H(s)$, of the chosen length.

In this paper, H has 256-bit long outputs. Indeed, such length is short enough to make the system efficient and long enough to make the system secure. For instance, we want H to be *collision-resilient*. That is, it should be hard to find two different strings x and y such that $H(x) = H(y)$. When H is a random oracle with 256-bit long outputs, finding any such pair of strings is indeed difficult. (Trying at random, and relying on the birthday paradox, would require $2^{256/2} = 2^{128}$ trials.)

Digital Signing. Digital signatures allow users to to authenticate information to each other without sharing any sharing any secret keys. A *digital signature scheme* consists of three fast algorithms: a probabilistic *key generator* G, a *signing algorithm* S, and a *verification algorithm* V.

Given a security parameter k, a sufficiently high integer, a user i uses G to produce a pair of k-bit keys (i.e., strings): a "public" key pk_i and a matching "secret" signing key sk_i. Crucially, a public key does not "betray" its corresponding secret key. That is, even given knowledge of pk_i, no one other than i is able to compute sk_i in less than astronomical time.

User i uses sk_i to digitally sign messages. For each possible message (binary string) m, i first hashes m and then runs algorithm S on inputs $H(m)$ and sk_i so as to produce the k-bit string

$$sig_{pk_i}(m) \triangleq S(H(m), sk_i) \ .^3$$

[3]Since H is collision-resilient it is practically impossible that, by signing m one "accidentally signs" a different message m'.

The binary string $sig_{pk_i}(m)$ is referred to as i's digital signature of m (relative to pk_i), and can be more simply denoted by $sig_i(m)$, when the public key pk_i is clear from context.

Everyone knowing pk_i can use it to verify the digital signatures produced by i. Specifically, on inputs (a) the public key pk_i of a player i, (b) a message m, and (c) a string s, that is, i's alleged digital signature of the message m, the verification algorithm V outputs either YES or NO.

The properties we require from a digital signature scheme are:

1. *Legitimate signatures are always verified:* If $s = sig_i(m)$, then $V(pk_i, m, s) = YES$; and
2. *Digital signatures are hard to forge:* Without knowledge of sk_i the time to find a string s such that $V(pk_i, m, s) = YES$, for a message m never signed by i, is astronomically long.

   (Following the strong security requirement of Goldwasser, Micali, and Rivest [17], this is true even if one can obtain the signature of any other message.)

Accordingly, to prevent anyone else from signing messages on his behalf, a player i must keep his signing key sk_i secret (hence the term "secret key"), and to enable anyone to verify the messages he does sign, i has an interest in publicizing his key pk_i (hence the term "public key").

In general, a message m is not retrievable from its signature $sig_i(m)$. In order to virtually deal with digital signatures that satisfy the conceptually convenient *"retrievability" property* (i.e., to guarantee that the signer and the message are easily computable from a signature, we define

$$SIG_{pk_i}(m) = (i, m, sig_{pk_i}(m)) \quad \text{and} \quad SIG_i(m) = (i, m, sig_i(m)), \text{ if } pk_i \text{ is clear.}$$

Unique Digital Signing. We also consider digital signature schemes (G, S, V) satisfying the following additional property.

3. *Uniqueness.* It is hard to find strings pk', m, s, and s' such that
$$s \neq s' \quad \text{and} \quad V(pk', m, s) = V(pk', m, s') = 1.$$

   (Note that the uniqueness property holds also for strings pk' that are not legitimately generated public keys. In particular, however, the uniqueness property implies that, if one used the specified key generator G to compute a public key pk together with a matching secret key sk, and thus knew sk, it would be essentially impossible also for him to find two different digital signatures of a same message relative to pk.)

Remarks

- From Unique signatures to verifiable random functions. Relative to a digital signature scheme with the uniqueness property, the mapping $m \to H(sig_i(m))$ associates to each possible string m, a unique, randomly selected, 256-bit string, and the correctness of this mapping can be proved given the signature $sig_i(m)$.

  That is, ideal hashing and digital signature scheme satisfying the uniqueness property essentially provide an elementary implementation of a *verifiable random function*, as introduced and by Micali, Rabin, and Vadhan [27]. (Their original implementation was necessarily more complex, since they did not rely on ideal hashing.)

- THREE DIFFERENT NEEDS FOR DIGITAL SIGNATURES. In Algorand, a user i relies on digital signatures for

  (1) *Authenticating i's own payments.* In this application, keys can be "long-term" (i.e., used to sign many messages over a long period of time) and come from a ordinary signature scheme.

  (2) *Generating credentials proving that i is entitled to act at some step s of a round r.* Here, keys can be long-term, but must come from a scheme satisfying the uniqueness property.

  (3) *Authenticating the message i sends in each step in which he acts.* Here, keys must be ephemeral (i.e., destroyed after their first use), but can come from an ordinary signature scheme.

- A SMALL-COST SIMPLIFICATION. For simplicity, we envision each user i to have a single long-term key. Accordingly, such a key must come from a signature scheme with the uniqueness property. Such simplicity has a small computational cost. Typically, in fact, unique digital signatures are slightly more expensive to produce and verify than ordinary signatures.

2.2 The Idealized Public Ledger

Algorand tries to mimic the following payment system, based on an *idealized public ledger*.

1. *The Initial Status.* Money is associated with individual public keys (privately generated and owned by users). Letting $pk_1, \ldots, pk_j$ be the initial public keys and $a_1, \ldots, a_j$ their respective initial amounts of money units, then the *initial status* is

   $$S_0 = (pk_1, a_1), \ldots, (pk_j, a_j) \ ,$$

   which is assumed to be common knowledge in the system.

2. *Payments.* Let pk be a public key currently having $a \geq 0$ money units, pk' another public key, and a' a non-negative number no greater than a. Then, a (valid) payment $\wp$ is a digital signature, relative to pk, specifying the transfer of a' monetary units from pk to pk', together with some additional information. In symbols,

   $$\wp = SIG_{pk}(pk, pk', a', I, H(\mathcal{I})),$$

   where I represents any additional information deemed useful but not sensitive (e.g., time information and a payment identifier), and $\mathcal{I}$ any additional information deemed sensitive (e.g., the reason for the payment, possibly the identities of the owners of pk and the pk', and so on).

   We refer to pk (or its owner) as the *payer*, to each pk' (or its owner) as a *payee*, and to a' as the *amount* of the payment $\wp$.

   Free Joining Via Payments. Note that users may join the system whenever they want by generating their own public/secret key pairs. Accordingly, the public key pk' that appears in the payment $\wp$ above may be a newly generated public key that had never "owned" any money before.

3. *The Magic Ledger.* In the Idealized System, all payments are valid and appear in a tamper-proof list L of sets of payments "posted on the sky" for everyone to see:

   $$L = PAY^1, PAY^2, \ldots,$$

Each block PAY^{r+1} consists of the set of all payments made since the appearance of block PAY^r. In the ideal system, a new block appears after a fixed (or finite) amount of time.

Discussion.

- *More General Payments and Unspent Transaction Output.* More generally, if a public key pk owns an amount a, then a valid payment $\wp$ of pk may transfer the amounts $a'_1, a'_2, \ldots$, respectively to the keys $pk'_1, pk'_2, \ldots$, so long as $\sum_j a'_j \leq a$.

  In Bitcoin and similar systems, the money owned by a public key pk is segregated into separate amounts, and a payment $\wp$ made by pk must transfer such a segregated amount a in its entirety. If pk wishes to transfer only a fraction $a' < a$ of a to another key, then it must also transfer the balance, the *unspent transaction output*, to another key, possibly pk itself.

  Algorand also works with keys having segregated amounts. However, in order to focus on the novel aspects of Algorand, it is conceptually simpler to stick to our simpler forms of payments and keys having a single amount associated to them.

- *Current Status.* The Idealized Scheme does not directly provide information about the current status of the system (i.e., about how many money units each public key has). This information is deducible from the Magic Ledger.

  In the ideal system, an active user continually stores and updates the latest status information, or he would otherwise have to reconstruct it, either from scratch, or from the last time he computed it. (In the next version of this paper, we shall augment Algorand so as to enable its users to reconstruct the current status in an efficient manner.)

- *Security and "Privacy".* Digital signatures guarantee that no one can forge a payment by another user. In a payment $\wp$, the public keys and the amount are not hidden, but the sensitive information $\mathcal{I}$ is. Indeed, only $H(\mathcal{I})$ appears in $\wp$, and since H is an ideal hash function, $H(\mathcal{I})$ is a random 256-bit value, and thus there is no way to figure out what $\mathcal{I}$ was better than by simply guessing it. Yet, to prove what $\mathcal{I}$ was (e.g., to prove the reason for the payment) the payer may just reveal $\mathcal{I}$. The correctness of the revealed $\mathcal{I}$ can be verified by computing $H(\mathcal{I})$ and comparing the resulting value with the last item of $\wp$. In fact, since H is *collision resilient,* it is hard to find a second value $\mathcal{I}'$ such that $H(\mathcal{I}) = H(\mathcal{I}')$.

2.3 Basic Notions and Notations

Keys, Users, and Owners Unless otherwise specified, each public key ("key" for short) is long-term and relative to a digital signature scheme with the uniqueness property. A public key i joins the system when another public key j already in the system makes a payment to i.

For color, we personify keys. We refer to a key i as a "he", say that i is honest, that i sends and receives messages, etc. *User* is a synonym for key. When we want to distinguish a key from the person to whom it belongs, we respectively use the term "digital key" and "owner".

Permissionless and Permissioned Systems. A system is *permissionless*, if a digital key is free to join at any time and an owner can own multiple digital keys; and its *permissioned*, otherwise.

Unique Representation Each object in Algorand has a unique representation. In particular, each set $\{(x, y, z, \ldots) : x \in X, y \in Y, z \in Z, \ldots\}$ is ordered in a pre-specified manner: e.g., first lexicographically in x, then in y, etc.

Same-Speed Clocks There is no global clock: rather, each user has his own clock. User clocks need not be synchronized in any way. We assume, however, that they all have the same speed.

For instance, when it is 12pm according to the clock of a user i, it may be 2:30pm according to the clock of another user j, but when it will be 12:01 according to i's clock, it will 2:31 according to j's clock. That is, "one minute is the same (sufficiently, essentially the same) for every user".

Rounds Algorand is organized in logical units, $r = 0, 1, \ldots$, called *rounds*.

We consistently use superscripts to indicate rounds. To indicate that a non-numerical quantity Q (e.g., a string, a public key, a set, a digital signature, etc.) refers to a round r, we simply write Q^r. Only when Q is a genuine number (as opposed to a binary string interpretable as a number), do we write $Q^{(r)}$, so that the symbol r could not be interpreted as the exponent of Q.

At (the start of a) round $r > 0$, the set of all public keys is PK^r, and the system status is

$$S^r = \left\{ \left(i, a_i^{(r)}, \ldots \right) : i \in PK^r \right\},$$

where $a_i^{(r)}$ is the amount of money available to the public key i. Note that PK^r is deducible from S^r, and that S^r may also specify other components for each public key i.

For round 0, PK^0 is the set of *initial public keys*, and S^0 is *the initial status*. Both PK^0 and S^0 are assumed to be common knowledge in the system. For simplicity, at the start of round r, so are $PK^1, \ldots, PK^r$ and $S^1, \ldots, S^r$.

In a round r, the system status transitions from S^r to S^{r+1}: symbolically,

$$\text{Round } r\text{: } S^r \longrightarrow S^{r+1}.$$

Payments In Algorand, the users continually make payments (and disseminate them in the way described in subsection 2.7). A payment $\wp$ of a user $i \in PK^r$ has the same format and semantics as in the Ideal System. Namely,

$$\wp = SIG_i(i, i', a, I, H(\mathcal{I})) \ .$$

Payment $\wp$ is *individually valid at a round* r (is a *round-*r *payment*, for short) if (1) its amount a is less than or equal to $a_i^{(r)}$, and (2) it does not appear in any official payset $PAY^{r'}$ for $r' < r$. (As explained below, the second condition means that $\wp$ has not already become effective.

A set of round-r payments of i is *collectively valid* if the sum of their amounts is at most $a_i^{(r)}$.

Paysets A round-r *payset* $\mathcal{P}$ is a set of round-r payments such that, for each user i, the payments of i in $\mathcal{P}$ (possibly none) are collectively valid. The set of all round-r paysets is $\mathbb{PAY}(r)$. A round-r payset $\mathcal{P}$ is *maximal* if no superset of $\mathcal{P}$ is a round-r payset.

We actually suggest that a payment $\wp$ also specifies a round ρ, $\wp = SIG_i(\rho, i, i', a, I, H(\mathcal{I}))$, and cannot be valid at any round outside $[\rho, \rho + k]$, for some fixed non-negative integer k.[4]

[4]This simplifies checking whether $\wp$ has become "effective" (i.e., it simplifies determining whether some payset PAY^r contains $\wp$. When $k = 0$, if $\wp = SIG_i(r, i, i', a, I, H(\mathcal{I}))$, and $\wp \notin PAY^r$, then i must re-submit $\wp$.

Official Paysets For every round r, Algorand publicly selects (in a manner described later on) a single (possibly empty) payset, PAY^r, the round's *official payset*. (Essentially, PAY^r represents the round-r payments that have *"actually"* happened.)

As in the Ideal System (and Bitcoin), (1) the only way for a new user j to enter the system is to be the recipient of a payment belonging to the official payset PAY^r of a given round r; and (2) PAY^r determines the status of the next round, S^{r+1}, from that of the current round, S^r. Symbolically,

$$PAY^r : S^r \longrightarrow S^{r+1}.$$

Specifically,

1. the set of public keys of round $r+1$, PK^{r+1}, consists of the union of PK^r and the set of all payee keys that appear, for the first time, in the payments of PAY^r; and
2. the amount of money $a_i^{(r+1)}$ that a user i owns in round $r+1$ is the sum of $a_i(r)$ —i.e., the amount of money i owned in the previous round (0 if $i \notin PK^r$)— and the sum of amounts paid to i according to the payments of PAY^r.

In sum, as in the Ideal System, each status S^{r+1} is deducible from the previous payment history:

$$PAY^0, \ldots, PAY^r.$$

2.4 Blocks and Proven Blocks

In $Algorand_0$, the block B^r corresponding to a round r specifies: r itself; the set of payments of round r, PAY^r; a quantity Q^r, to be explained, and the hash of the previous block, $H(B^{r-1})$. Thus, starting from some fixed block B^0, we have a traditional blockchain:

$$B^1 = (1, PAY^1, Q^0, H(B^0)), \quad B^2 = (2, PAY^2, Q^1, H(B^1)), \quad B^3 = (3, PAY^3, Q^2, H(B^2)), \quad \ldots$$

In Algorand, the authenticity of a block is actually vouched by a separate piece of information, a "block certificate" $CERT^r$, which turns B^r into a *proven block*, $\overline{B^r}$. The Magic Ledger, therefore, is implemented by the sequence of the proven blocks,

$$\overline{B^1}, \overline{B^2}, \ldots$$

Discussion As we shall see, $CERT^r$ consists of a set of digital signatures for $H(B^r)$, those of a majority of the members of SV^r, together with a proof that each of those members indeed belongs to SV^r. We could, of course, include the certificates $CERT^r$ in the blocks themselves, but find it conceptually cleaner to keep it separate.)

In Bitcoin each block must satisfy a special property, that is, must "contain a solution of a crypto puzzle", which makes block generation computationally intensive and forks both inevitable and not rare. By contrast, Algorand's blockchain has two main advantages: it is generated with minimal computation, and it will not fork with overwhelmingly high probability. Each block B^i is safely *final* as soon as it enters the blockchain.

2.5 Acceptable Failure Probability

To analyze the security of Algorand we specify the probability, F, with which we are willing to accept that something goes wrong (e.g., that a verifier set SV^r does not have an honest majority). As in the case of the output length of the cryptographic hash function H, also F is a parameter. But, as in that case, we find it useful to set F to a concrete value, so as to get a more intuitive grasp of the fact that it is indeed possible, in Algorand, to enjoy simultaneously sufficient security and sufficient efficiency. To emphasize that F is parameter that can be set as desired, in the first and second embodiments we respectively set

$$F = 10^{-12} \quad \text{and} \quad F = 10^{-18} \ .$$

Discussion Note that 10^{-12} is actually less than one in a trillion, and we believe that such a choice of F is adequate in our application. Let us emphasize that 10^{-12} is *not* the probability with which the Adversary can forge the payments of an honest user. All payments are digitally signed, and thus, if the proper digital signatures are used, the probability of forging a payment is far lower than 10^{-12}, and is, in fact, essentially 0. The bad event that we are willing to tolerate with probability F is that Algorand's blockchain *forks*. Notice that, with our setting of F and one-minute long rounds, a fork is expected to occur in Algorand's blockchain as infrequently as (roughly) once in 1.9 million years. By contrast, in Bitcoin, a forks occurs quite often.

A more demanding person may set F to a lower value. To this end, in our second embodiment we consider setting F to 10^{-18}. Note that, assuming that a block is generated every *second*, 10^{18} is the estimated number of seconds taken by the Universe so far: from the Big Bang to present time. Thus, with $F = 10^{-18}$, if a block is generated in a second, one should expect for the age of the Universe to see a fork.

2.6 The Adversarial Model

Algorand is designed to be secure in a very adversarial model. Let us explain.

Honest and Malicious Users A user is *honest* if he follows all his protocol instructions, and is perfectly capable of sending and receiving messages. A user is *malicious* (i.e., *Byzantine*, in the parlance of distributed computing) if he can deviate arbitrarily from his prescribed instructions.

The Adversary The *Adversary* is an efficient (technically polynomial-time) algorithm, personified for color, who can *immediately* make malicious *any user* he wants, at *any time* he wants (subject only to an upperbound to the number of the users he can corrupt).

The Adversary totally controls and perfectly coordinates all malicious users. He takes all actions on their behalf, including receiving and sending all their messages, and can let them deviate from their prescribed instructions in arbitrary ways. Or he can simply isolate a corrupted user sending and receiving messages. Let us clarify that no one else automatically learns that a user i is malicious, although i's maliciousness may transpire by the actions the Adversary has him take.

This powerful adversary however,

- Does not have unbounded computational power and cannot successfully forge the digital signature of an honest user, except with negligible probability; and

- Cannot interfere in any way with the messages exchanges among honest users.

Furthermore, his ability to attack honest users is bounded by one of the following assumption.

Honesty Majority of Money We consider a continuum of Honest Majority of Money (HMM) assumptions: namely, for each non-negative integer k and real $h > 1/2$,

> $HHM_k > h$: *the honest users in every round r owned a fraction greater than h of all money in the system at round $r - k$.*

Discussion. Assuming that all malicious users perfectly coordinate their actions (as if controlled by a single entity, the Adversary) is a rather pessimistic hypothesis. Perfect coordination among too many individuals is difficult to achieve. Perhaps coordination only occurs within separate groups of malicious players. But, since one cannot be sure about the level of coordination malicious users may enjoy, we'd better be safe than sorry.

Assuming that the Adversary can secretly, dynamically, and immediately corrupt users is also pessimistic. After all, realistically, taking full control of a user's operations should take some time.

The assumption $HMM_k > h$ implies, for instance, that, if a round (on average) is implemented in one minute, then, the majority of the money at a given round will remain in honest hands for at least two hours, if $k = 120$, and at least one week, if $k = 10,000$.

Note that the HMM assumptions and the previous Honest Majority of Computing Power assumptions are related in the sense that, since computing power can be bought with money, if malicious users own most of the money, then they can obtain most of the computing power.

2.7 The Communication Model

We envisage message propagation —i.e., "peer-to-peer gossip"[5]— to be the only means of communication.

Temporary Assumption: Timely Delivery of Messages in the Entire Network. For most part of this paper we assume that every propagated message reaches almost all honest users in a timely fashion. We shall remove this assumption in Section 10, where we deal with network partitions, either naturally occurring or adversarially induced. (As we shall see, we only assume timely delivery of messages within each connected component of the network.)

One concrete way to capture timely delivery of propagated messages (in the entire network) is the following:

> *For all* reachability $\rho > 95\%$ *and* message size $\mu \in \mathbb{Z}_+$, *there exists* $\lambda_{\rho,\mu}$ *such that,*
>
> *if a honest user propagates μ-byte message m at time t,*
>
> *then m reaches, by time $t + \lambda_{\rho,\mu}$, at least a fraction ρ of the honest users.*

[5] Essentially, as in Bitcoin, when a user propagates a message m, every active user i receiving m for the first time, randomly and independently selects a suitably small number of active users, his *"neighbors"*, to whom he forwards m, possibly until he receives an acknowledgement from them. The propagation of m terminates when no user receives m for the first time.

The above property, however, cannot support our Algorand protocol, without explicitly and separately envisaging a mechanism to obtain the latest blockchain —by another user/depository/etc. In fact, to construct a new block B^r not only should a proper set of verifiers timely receive round-r messages, but also the messages of previous rounds, so as to know B^{r-1} and all other previous blocks, which is necessary to determine whether the payments in B^r are valid. The following assumption instead suffices.

Message Propagation (MP) Assumption: *For all $\rho > 95\%$ and $\mu \in \mathbb{Z}_+$, there exists $\lambda_{\rho,\mu}$ such that, for all times t and all μ-byte messages m propagated by an honest user before $t - \lambda_{\rho,\mu}$, m is received, by time t, by at least a fraction ρ of the honest users.*

Protocol *Algorand′* actually instructs each of a small number of users (i.e., the verifiers of a given step of a round in *Algorand′*, to propagate a separate message of a (small) prescribed size, and we need to bound the time required to fulfill these instructions. We do so by enriching the MP assumption as follows.

For all n, $\rho > 95\%$, and $\mu \in \mathbb{Z}_+$, there exists $\lambda_{n,\rho,\mu}$ such that, for all times t and all μ-byte messages $m_1, \ldots, m_n$, each propagated by an honest user before $t - \lambda_{n,\rho,\mu}$, $m_1, \ldots, m_n$ are received, by time t, by at least a fraction ρ of the honest users.

Note

- The above assumption is deliberately simple, but also stronger than needed in our paper.[6]
- For simplicity, we assume $\rho = 1$, and thus drop mentioning ρ.
- We pessimistically assume that, provided he does not violate the MP assumption, the Adversary totally controls the delivery of all messages. In particular, without being noticed by the honest users, the Adversary he can arbitrarily decide which honest player receives which message when, and arbitrarily accelerate the delivery of any message he wants.[7]

3 The BA Protocol $BA^\star$ in a Traditional Setting

As already emphasized, Byzantine agreement is a key ingredient of Algorand. Indeed, it is through the use of such a BA protocol that Algorand is unaffected by forks. However, to be secure against our powerful Adversary, Algorand must rely on a BA protocol that satisfies the new player-replaceability constraint. In addition, for Algorand to be efficient, such a BA protocol must be very efficient.

BA protocols were first defined for an idealized communication model, *synchronous complete networks* (SC networks). Such a model allows for a simpler design and analysis of BA protocols.

[6] Given the honest percentage h and the acceptable failure probability F, Algorand computes an upperbound, N, to the maximum number of member of verifiers in a step. Thus, the MP assumption need only hold for $n \leq N$.

In addition, as stated, the MP assumption holds no matter how many other messages may be propagated alongside the m_j's. As we shall see, however, in Algorand messages at are propagated in essentially non-overlapping time intervals, during which either a single block is propagated, or at most N verifiers propagate a small (e.g., 200B) message. Thus, we could restate the MP assumption in a weaker, but also more complex, way.

[7] For instance, he can immediately learn the messages sent by honest players. Thus, a malicious user i', who is asked to propagate a message simultaneously with a honest user i, can always choose his own message m' based on the message m actually propagated by i. This ability is related to *rushing*, in the parlance of distributed-computation literature.

Accordingly, in this section, we introduce a new BA protocol, $BA^\star$, for SC networks and ignoring the issue of player replaceability altogether. The protocol $BA^\star$ is a contribution of separate value. Indeed, it is the most efficient cryptographic BA protocol for SC networks known so far.

To use it within our Algorand protocol, we modify $BA^\star$ a bit, so as to account for our different communication model and context, but make sure, in section X, to highlight how $BA^\star$ is used within our actual protocol *Algorand′*.

We start by recalling the model in which $BA^\star$ operates and the notion of a Byzantine agreement.

3.1 Synchronous Complete Networks and Matching Adversaries

In a SC network, there is a common clock, ticking at each integral times $r = 1, 2, \ldots$

At each even time click r, each player i instantaneously and simultaneously sends a single message $m_{i,j}^r$ (possibly the empty message) to each player j, including himself. Each $m_{i,j}^r$ is received at time click $r + 1$ by player j, together with the identity of the sender i.

Again, in a communication protocol, a player is *honest* if he follows all his prescribed instructions, and *malicious* otherwise. All malicious players are totally controlled and perfectly coordinated by the Adversary, who, in particular, immediately receives all messages addressed to malicious players, and chooses the messages they send.

The Adversary can immediately make malicious any honest user he wants at any odd time click he wants, subject only to a possible upperbound t to the number of malicious players. That is, the Adversary "cannot interfere with the messages already sent by an honest user i", which will be delivered as usual.

The Adversary also has the additional ability to see instantaneously, at each even round, the messages that the currently honest players send, and instantaneously use this information to choose the messages the malicious players send at the same time tick.

Remarks

- *Adversary Power.* The above setting is very adversarial. Indeed, in the Byzantine agreement literature, many settings are less adversarial. However, some more adversarial settings have also been considered, where the Adversary, after seeing the messages sent by an honest player i at a given time click r, has the ability to erase all these messages from the network, immediately corrupt i, choose the message that the now malicious i sends at time click r, and have them delivered as usual. The envisaged power of the Adversary matches that he has in our setting.

- *Physical Abstraction.* The envisaged communication model abstracts a more physical model, in which each pair of players (i, j) is linked by a separate and private communication line $l_{i,j}$. That is, no one else can inject, interfere with, or gain information about the messages sent over $l_{i,j}$. The only way for the Adversary to have access to $l_{i,j}$ is to corrupt either i or j.

- *Privacy and Authentication.* In SC networks message privacy and authentication are guaranteed by assumption. By contrast, in our communication network, where messages are propagated from peer to peer, authentication is guaranteed by digital signatures, and privacy is non-existent. Thus, to adopt protocol $BA^\star$ to our setting, each message exchanged should be digitally signed (further identifying the state at which it was sent). Fortunately, the BA protocols that we consider using in Algorand do not require message privacy.

3.2 The Notion of a Byzantine Agreement

The notion of Byzantine agreement was introduced by Pease Shostak and Lamport [31] for the *binary* case, that is, when every initial value consists of a bit. However, it was quickly extended to arbitrary initial values. (See the surveys of Fischer [16] and Chor and Dwork [10].) By a BA protocol, we mean an arbitrary-value one.

Definition 3.1. *In a synchronous network, let $\mathcal{P}$ be a n-player protocol, whose player set is common knowledge among the players, t a positive integer such that $n \geq 2t+1$. We say that $\mathcal{P}$ is an* arbitrary-value *(respectively,* binary*)* (n,t)-Byzantine agreement protocol *with* soundness $\sigma \in (0,1)$ *if, for every set of values V not containing the special symbol $\perp$ (respectively, for $V = \{0,1\}$), in an execution in which at most t of the players are malicious and in which every player i starts with an* initial value $v_i \in V$*, every honest player j halts with probability 1, outputting a value $out_i \in V \cup \{\perp\}$ so as to satisfy, with probability at least σ, the following two conditions:*

1. Agreement: *There exists $out \in V \cup \{\perp\}$ such that $out_i = out$ for all honest players i.*
2. Consistency: *if, for some value $v \in V$, $v_i = v$ for all honest players, then $out = v$.*

We refer to out as $\mathcal{P}$'s output*, and to each out_i as* player i's output*.*

3.3 The BA Notation

In our BA protocols, a player is required to count how many players sent him a given message in a given step. Accordingly, for each possible value v that might be sent,

$$\#_i^s(v)$$

(or just $\#_i(v)$ when s is clear) is the number of players j from which i has received v in step s.

Recalling that a player i receives exactly one message from each player j, if the number of players is n, then, for all i and s, $\sum_v \#_i^s(v) = n$.

3.4 The Binary BA Protocol $BBA^\star$

In this section we present a new *binary* BA protocol, $BBA^\star$, which relies on the honesty of more than two thirds of the players and is very fast: no matter what the malicious players might do, each execution of its main loop brings the players into agreement with probability 1/3.

Each player has his own public key of a digital signature scheme satisfying the unique-signature property. Since this protocol is intended to be run on synchronous complete network, there is no need for a player i to sign each of his messages.

Digital signatures are used to generate a sufficiently common random bit in Step 3. (In Algorand, digital signatures are used to authenticate all other messages as well.)

The protocol requires a minimal set-up: a common random string r, independent of the players' keys. (In Algorand, r is actually replaced by the quantity Q^r.)

Protocol $BBA^\star$ is a 3-step loop, where the players repeatedly exchange Boolean values, and different players may exit this loop at different times. A player i exits this loop by propagating, at some step, either a special value $0*$ or a special value $1*$, thereby instructing all players to "pretend" they respectively receive 0 and 1 from i in all future steps. (Alternatively said: assume

that the last message received by a player j from another player i was a bit b. Then, in any step in which he does not receive any message from i, j acts as if i sent him the bit b.)

The protocol uses a counter γ, representing how many times its 3-step loop has been executed. At the start of $BBA^\star$, $\gamma = 0$. (One may think of γ as a global counter, but it is actually increased by each individual player every time that the loop is executed.)

There are $n \geq 3t+1$, where t is the maximum possible number of malicious players. A binary string x is identified with the integer whose binary representation (with possible leadings 0s) is x; and $\texttt{lsb}(x)$ denotes the least significant bit of x.

PROTOCOL $BBA^\star$

(COMMUNICATION) STEP 1. [Coin-Fixed-To-0 Step] *Each player i sends b_i.*

1.1 If $\#_i^1(0) \geq 2t+1$, then i sets $b_i = 0$, sends $0*$, outputs $out_i = 0$, and HALTS.

1.2 If $\#_i^1(1) \geq 2t+1$, then, then i sets $b_i = 1$.

1.3 Else, i sets $b_i = 0$.

(COMMUNICATION) STEP 2. [Coin-Fixed-To-1 Step] *Each player i sends b_i.*

2.1 If $\#_i^2(1) \geq 2t+1$, then i sets $b_i = 1$, sends $1*$, outputs $out_i = 1$, and HALTS.

2.2 If $\#_i^2(0) \geq 2t+1$, then i set $b_i = 0$.

2.3 Else, i sets $b_i = 1$.

(COMMUNICATION) STEP 3. [Coin-Genuinely-Flipped Step] *Each player i sends b_i and $SIG_i(r, \gamma)$.*

3.1 If $\#_i^3(0) \geq 2t+1$, then i sets $b_i = 0$.

3.2 If $\#_i^3(1) \geq 2t+1$, then i sets $b_i = 1$.

3.3 Else, letting $S_i = \{j \in N$ who have sent i a proper message in this step 3 $\}$, i sets $b_i = c \triangleq \texttt{lsb}(\min_{j \in S_i} H(SIG_i(r, \gamma)))$; increases γ_i by 1; and returns to Step 1.

Theorem 3.1. *Whenever $n \geq 3t+1$, $BBA^\star$ is a binary (n,t)-BA protocol with soundness 1.*

A proof of Theorem 3.1 is given in [26]. Its adaptation to our setting, and its player-replaceability property are novel.

Historical Remark Probabilistic binary BA protocols were first proposed by Ben-Or in asynchronous settings [7]. Protocol $BBA^\star$ is a novel adaptation, to our public-key setting, of the binary BA protocol of Feldman and Micali [15]. Their protocol was the first to work in an expected constant number of steps. It worked by having the players themselves implement a *common coin*, a notion proposed by Rabin, who implemented it via an external trusted party [32].

3.5 Graded Consensus and the Protocol *GC*

Let us recall, for arbitrary values, a notion of consensus much weaker than Byzantine agreement.

Definition 3.2. *Let $\mathcal{P}$ be a protocol in which the set of all players is common knowledge, and each player i privately knows an arbitrary* initial value v'_i.

We say that $\mathcal{P}$ is an (n,t)-graded consensus protocol *if, in every execution with n players, at most t of which are malicious, every honest player i halts outputting a* value-grade *pair (v_i, g_i), where $g_i \in \{0,1,2\}$, so as to satisfy the following three conditions:*

1. *For all honest players i and j, $|g_i - g_j| \leq 1$.*
2. *For all honest players i and j, $g_i, g_j > 0 \Rightarrow v_i = v_j$.*
3. *If $v'_1 = \cdots = v'_n = v$ for some value v, then $v_i = v$ and $g_i = 2$ for all honest players i.*

Historical Note The notion of a graded consensus is simply derived from that of a *graded broadcast*, put forward by Feldman and Micali in [15], by strengthening the notion of a *crusader agreement*, as introduced by Dolev [12], and refined by Turpin and Coan [33].[8]

In [15], the authors also provided a 3-step (n,t)-graded broadcasting protocol, *gradecast*, for $n \geq 3t+1$. A more complex (n,t)-graded-broadcasting protocol for $n > 2t+1$ has later been found by Katz and Koo [19].

The following two-step protocol *GC* consists of the last two steps of *gradecast*, expressed in our notation. To emphasize this fact, and to match the steps of protocol *Algorand′* of section 4.1, we respectively name 2 and 3 the steps of *GC*.

PROTOCOL *GC*

STEP 2. *Each player i sends v'_i to all players.*

STEP 3. *Each player i sends to all players the string x if and only if $\#_i^2(x) \geq 2t+1$.*

OUTPUT DETERMINATION. *Each player i outputs the pair (v_i, g_i) computed as follows:*

- *If, for some x, $\#_i^3(x) \geq 2t+1$, then $v_i = x$ and $g_i = 2$.*
- *If, for some x, $\#_i^3(x) \geq t+1$, then $v_i = x$ and $g_i = 1$.*
- *Else, $v_i = \perp$ and $g_i = 0$.*

Theorem 3.2. *If $n \geq 3t+1$, then GC is a (n,t)-graded broadcast protocol.*

The proof immediately follows from that of the protocol *gradecast* in [15], and is thus omitted.[9]

[8] In essence, in a graded-broadcasting protocol, (a) the input of every player is the identity of a distinguished player, the *sender*, who has an arbitrary value v as an additional private input, and (b) the outputs must satisfy the same properties 1 and 2 of graded consensus, plus the following property 3′: *if the sender is honest, then $v_i = v$ and $g_i = 2$ for all honest player i.*

[9] Indeed, in their protocol, in step 1, the sender sends his own private value v to all players, and each player i lets v'_i consist of the value he has actually received from the sender in step 1.

3.6 The Protocol $BA^\star$

We now describe the arbitrary-value BA protocol $BA^\star$ via the binary BA protocol $BBA^\star$ and the graded-consensus protocol GC. Below, the initial value of each player i is v'_i.

PROTOCOL $BA^\star$

STEPS 1 AND 2. *Each player i executes GC, on input v'_i, so as to compute a pair (v_i, g_i).*

STEP 3, ... *Each player i executes $BBA^\star$ —with initial input 0, if $g_i = 2$, and 1 otherwise— so as to compute the bit out_i.*

OUTPUT DETERMINATION. *Each player i outputs v_i, if $out_i = 0$, and $\perp$ otherwise.*

Theorem 3.3. *Whenever $n \geq 3t + 1$, $BA^\star$ is a (n, t)-BA protocol with soundness 1.*

Proof. We first prove Consistency, and then Agreement.

PROOF OF CONSISTENCY. Assume that, for some value $v \in V$, $v'_i = v$. Then, by property 3 of graded consensus, after the execution of GC, all honest players output $(v, 2)$. Accordingly, 0 is the initial bit of all honest players in the end of the execution of $BBA^\star$. Thus, by the Agreement property of binary Byzantine agreement, at the end of the execution of $BA^\star$, $out_i = 0$ for all honest players. This implies that the output of each honest player i in $BA^\star$ is $v_i = v$. □

PROOF OF AGREEMENT. Since $BBA^\star$ is a binary BA protocol, either
(A) $out_i = 1$ for all honest player i, or
(B) $out_i = 0$ for all honest player i.
In case A, all honest players output $\perp$ in $BA^\star$, and thus Agreement holds. Consider now case B. In this case, in the execution of $BBA^\star$, the initial bit of at least one honest player i is 0. (Indeed, if initial bit of all honest players were 1, then, by the Consistency property of $BBA^\star$, we would have $out_j = 1$ for all honest j.) Accordingly, after the execution of GC, i outputs the pair $(v, 2)$ for some value v. Thus, by property 1 of graded consensus, $g_j > 0$ for all honest players j. Accordingly, by property 2 of graded consensus, $v_j = v$ for all honest players j. This implies that, at the end of $BA^\star$, every honest player j outputs v. Thus, Agreement holds also in case B. □

Since both Consistency and Agreement hold, $BA^\star$ is an arbitrary-value BA protocol. ■

Historical Note Turpin and Coan were the first to show that, for $n \geq 3t+1$, any binary (n, t)-BA protocol can be converted to an arbitrary-value (n, t)-BA protocol. The reduction arbitrary-value Byzantine agreement to binary Byzantine agreement via graded consensus is more modular and cleaner, and simplifies the analysis of our Algorand protocol *Algorand′*.

Generalizing $BA^\star$ for use in Algorand Algorand works even when all communication is via gossiping. However, although presented in a traditional and familiar communication network, so as to enable a better comparison with the prior art and an easier understanding, protocol $BA^\star$ works also in gossiping networks. In fact, in our detailed embodiments of Algorand, we shall present it directly for gossiping networks. We shall also point out that it satisfies the player replaceability property that is crucial for Algorand to be secure in the envisaged very adversarial model.

Any BA player-replaceable protocol working in a gossiping communication network can be securely employed within the inventive Algorand system. In particular, Micali and Vaikunthanatan have extended $BA^\star$ to work very efficiently also with a simple majority of honest players. That protocol too could be used in Algorand.

4 Two Embodiments of Algorand

As discussed, at a very high level, a round of Algorand ideally proceeds as follows. First, a randomly selected user, the leader, proposes and circulates a new block. (This process includes initially selecting a few potential leaders and then ensuring that, at least a good fraction of the time, a single common leader emerges.) Second, a randomly selected committee of users is selected, and reaches Byzantine agreement on the block proposed by the leader. (This process includes that each step of the BA protocol is run by a separately selected committee.) The agreed upon block is then digitally signed by a given threshold (T_H) of committee members. These digital signatures are circulated so that everyone is assured of which is the new block. (This includes circulating the credential of the signers, and authenticating just the hash of the new block, ensuring that everyone is guaranteed to learn the block, once its hash is made clear.)

In the next two sections, we present two embodiments of Algorand, $Algorand'_1$ and $Algorand'_2$, that work under a majority-of-honest-users assumption. In Section 8 we show how to adopts these embodiments to work under a honest-majority-of-money assumption.

$Algorand'_1$ only envisages that $> 2/3$ of the committee members are honest. In addition, in $Algorand'_1$, the number of steps for reaching Byzantine agreement is capped at a suitably high number, so that agreement is guaranteed to be reached with overwhelming probability within a fixed number of steps (but potentially requiring longer time than the steps of $Algorand'_2$). In the remote case in which agreement is not yet reached by the last step, the committee agrees on the empty block, which is always valid.

$Algorand'_2$ envisages that the number of honest members in a committee is always greater than or equal to a fixed threshold t_H (which guarantees that, with overwhelming probability, at least $2/3$ of the committee members are honest). In addition, $Algorand'_2$ allows Byzantine agreement to be reached in an arbitrary number of steps (but potentially in a shorter time than $Algorand'_1$).

It is easy to derive many variants of these basic embodiments. In particular, it is easy, given $Algorand'_2$, to modify $Algorand'_1$ so as to enable to reach Byzantine agreement in an arbitrary number of steps.

Both embodiments share the following common core, notations, notions, and parameters.

4.1 A Common Core

Objectives Ideally, for each round r, Algorand would satisfy the following properties:

1. *Perfect Correctness.* All honest users agree on the same block B^r.

2. *Completeness 1.* With probability 1, the payset of B^r, PAY^r, is maximal.[10]

[10] Because paysets are defined to contain valid payments, and honest users to make only valid payments, a maximal PAY^r contains the "currently outstanding" payments of all honest users.

Of course, guaranteeing perfect correctness alone is trivial: everyone always chooses the official payset PAY^r to be empty. But in this case, the system would have completeness 0. Unfortunately, guaranteeing both perfect correctness and completeness 1 is not easy in the presence of malicious users. Algorand thus adopts a more realistic objective. Informally, letting h denote the percentage of users who are honest, $h > 2/3$, the goal of Algorand is

Guaranteeing, with overwhelming probability, perfect correctness and completeness close to h.

Privileging correctness over completeness seems a reasonable choice: payments not processed in one round can be processed in the next, but one should avoid *forks*, if possible.

Led Byzantine Agreement Perfect Correctness could be guaranteed as follows. At the start of round r, each user i constructs his own candidate block B_i^r, and then all users reach Byzantine agreement on one candidate block. As per our introduction, the BA protocol employed requires a 2/3 honest majority and is player replaceable. Each of its step can be executed by a small and randomly selected set of *verifiers*, who do not share any inner variables.

Unfortunately, this approach has no completeness guarantees. This is so, because the candidate blocks of the honest users are most likely totally different from each other. Thus, the ultimately agreed upon block might always be one with a non-maximal payset. In fact, it may always be the empty block, B_ε, that is, the block whose payset is empty. well be the default, empty one.

Algorand′ avoids this completeness problem as follows. First, a leader for round r, ℓ^r, is selected. Then, ℓ^r propagates his own candidate block, $B_{\ell^r}^r$. Finally, the users reach agreement on the block they actually receive from ℓ^r. Because, whenever ℓ^r is honest, Perfect Correctness and Completeness 1 both hold, *Algorand′* ensures that ℓ^r is honest with probability close to h. (When the leader is malicious, we do not care whether the agreed upon block is one with an empty payset. After all, a malicious leader ℓ^r might always maliciously choose $B_{\ell^r}^r$ to be the empty block, and then honestly propagate it, thus forcing the honest users to agree on the empty block.)

Leader Selection In Algorand's, the rth block is of the form $B^r = (r, PAY^r, Q^r, H(B^{r-1})$. As already mentioned in the introduction, the quantity Q^{r-1} is carefully constructed so as to be essentially non-manipulatable by our very powerful Adversary. (Later on in this section, we shall provide some intuition about why this is the case.) At the start of a round r, all users know the blockchain so far, $B^0, \ldots, B^{r-1}$, from which they deduce the set of users of every prior round: that is, $PK^1, \ldots, PK^{r-1}$. A *potential leader* of round r is a user i such that

$$.H\left(SIG_i\left(r, 1, Q^{r-1}\right)\right) \leq p \ .$$

Let us explain. Note that, since the quantity Q^{r-1} is part of block B^{r-1}, and the underlying signature scheme satisfies the uniqueness property, $SIG_i\left(r, 1, Q^{r-1}\right)$ is a binary string uniquely associated to i and r. Thus, since H is a random oracle, $H\left(SIG_i\left(r, 1, Q^{r-1}\right)\right)$ is a random 256-bit long string uniquely associated to i and r. The symbol "." in front of $H\left(SIG_i\left(r, 1, Q^{r-1}\right)\right)$ is the *decimal* (in our case, *binary*) *point*, so that $r_i \triangleq .H\left(SIG_i\left(r, 1, Q^{r-1}\right)\right)$ is the binary expansion of a random 256-bit number between 0 and 1 uniquely associated to i and r. Thus the probability that r_i is less than or equal to p is essentially p. (Our potential-leader selection mechanism has been inspired by the micro-payment scheme of Micali and Rivest [28].)

The probability p is chosen so that, with overwhelming (i.e., $1 - F$) probability, at least one potential verifier is honest. (If fact, p is chosen to be the smallest such probability.)

Note that, since i is the only one capable of computing his own signatures, he alone can determine whether he is a potential verifier of round 1. However, by revealing his own *credential*, $\sigma_i^r \triangleq SIG_i\left(r, 1, Q^{r-1}\right)$, i can prove to anyone to be a potential verifier of round r.

The leader ℓ^r is defined to be the potential leader whose hashed credential is smaller that the hashed credential of all other potential leader j: that is, $H(\sigma_{\ell^r}^{r,s}) \leq H(\sigma_j^{r,s})$.

Note that, since a malicious ℓ^r may not reveal his credential, the correct leader of round r may never be known, and that, barring improbable ties, ℓ^r is indeed the only leader of round r.

Let us finally bring up a last but important detail: a user i can be a potential leader (and thus the leader) of a round r only if he belonged to the system for at least k rounds. This guarantees the non-manipulatability of Q^r and all future Q-quantities. In fact, one of the potential leaders will actually determine Q^r.

Verifier Selection Each step $s > 1$ of round r is executed by a small set of verifiers, $SV^{r,s}$. Again, each verifier $i \in SV^{r,s}$ is randomly selected among the users already in the system k rounds before r, and again via the special quantity Q^{r-1}. Specifically, $i \in PK^{r-k}$ is a *verifier* in $SV^{r,s}$, if

$$.H\left(SIG_i\left(r, s, Q^{r-1}\right)\right) \leq p' \ .$$

Once more, only i knows whether he belongs to $SV^{r,s}$,but, if this is the case, he could prove it by exhibiting his credential $\sigma_i^{r,s} \triangleq H(SIG_i\left(r, s, Q^{r-1}\right))$. A verifier $i \in SV^{r,s}$ sends a message, $m_i^{r,s}$, in step s of round r, and this message includes his credential $\sigma_i^{r,s}$, so as to enable the verifiers f the nest step to recognize that $m_i^{r,s}$ is a legitimate step-s message.

The probability p' is chosen so as to ensure that, in $SV^{r,s}$, letting $\#good$ be the number of honest users and $\#bad$ the number of malicious users, with overwhelming probability the following two conditions hold.

For embodiment $Algorand'_1$:
(1) $\#good > 2 \cdot \#bad$ and
(2) $\#good + 4 \cdot \#bad < 2n$, where n is the expected cardinality of $SV^{r,s}$.

For embodiment $Algorand'_2$:
(1) $\#good > t_H$ and
(2) $\#good + 2\#bad < 2t_H$, where t_H is a specified threshold.

These conditions imply that, with sufficiently high probability, (a) in the last step of the BA protocol, there will be at least given number of honest players to digitally sign the new block B^r, (b) only one block per round may have the necessary number of signatures, and (c) the used BA protocol has (at each step) the required 2/3 honest majority.

Clarifying Block Generation If the round-r leader ℓ^r is honest, then the corresponding block is of the form

$$B^r = \left(r, PAY^r, SIG_{\ell^r}\left(Q^{r-1}\right), H\left(B^{r-1}\right)\right) \ ,$$

where the payset PAY^r is maximal. (recall that all paysets are, by definition, collectively valid.)

Else (i.e., if ℓ^r is malicious), B^r has one of the following two possible forms:

$$B^r = \left(r, PAY^r, SIG_i\left(Q^{r-1}\right), H\left(B^{r-1}\right)\right) \quad \text{and} \quad B^r = B_\varepsilon^r \triangleq \left(r, \emptyset, Q^{r-1}, H\left(B^{r-1}\right)\right) \ .$$

In the first form, PAY^r is a (non-necessarily maximal) payset and it may be $PAY^r = \emptyset$; and i is a potential leader of round r. (However, i may not be the leader ℓ^r. This may indeed happen if if ℓ^r keeps secret his credential and does not reveal himself.)

The second form arises when, in the round-r execution of the BA protocol, all honest players output the default value, which is the empty block B^r_ε in our application. (By definition, the possible outputs of a BA protocol include a default value, generically denoted by $\perp$. See section 3.2.)

Note that, although the paysets are empty in both cases, $B^r = \left(r, \emptyset, SIG_i\left(Q^{r-1}\right), H\left(B^{r-1}\right)\right)$ and B^r_ε are syntactically different blocks and arise in two different situations: respectively, "all went smoothly enough in the execution of the BA protocol", and "something went wrong in the BA protocol, and the default value was output".

Let us now intuitively describe how the generation of block B^r proceeds in round r of *Algorand′*. In the first step, each eligible player, that is, each player $i \in PK^{r-k}$, checks whether he is a potential leader. If this is the case, then i is asked, using of all the payments he has seen so far, and the current blockchain, $B^0, \ldots, B^{r-1}$, to secretly prepare a maximal payment set, PAY^r_i, and secretly assembles his candidate block, $B^r = \left(r, PAY^r_i, SIG_i\left(Q^{r-1}\right), H\left(B^{r-1}\right)\right)$. That,is, not only does he include in B^r_i, as its second component the just prepared payset, but also, as its third component, his own signature of Q^{r-1}, the third component of the last block, B^{r-1}. Finally, he propagate his round-r-step-1 message, $m^{r,1}_i$, which includes (a) his candidate block B^r_i, (b) his proper signature of his candidate block (i.e., his signature of the hash of B^r_i, and (c) his own credential $\sigma^{r,1}_i$, proving that he is indeed a potential verifier of round r.

(Note that, until an honest i produces his message $m^{r,1}_i$, the Adversary has no clue that i is a potential verifier. Should he wish to corrupt honest potential leaders, the Adversary might as well corrupt random honest players. However, once he sees $m^{r,1}_i$, since it contains i's credential, the Adversary knows and could corrupt i, but cannot prevent $m^{r,1}_i$, which is virally propagated, from reaching all users in the system.)

In the second step, each selected verifier $j \in SV^{r,2}$ tries to identify the leader of the round. Specifically, j takes the step-1 credentials, $\sigma^{r,1}_{i_1}, \ldots, \sigma^{r,1}_{i_n}$, contained in the proper step-1 message $m^{r,1}_i$ he has received; hashes all of them, that is, computes $H\left(\sigma^{r,1}_{i_1}\right), \ldots, H\left(\sigma^{r,1}_{i_n}\right)$; finds the credential, $\sigma^{r,1}_{\ell_j}$, whose hash is lexicographically minimum; and considers ℓ^r_j to be the leader of round r.

Recall that each considered credential is a digital signature of Q^{r-1}, that $SIG_i\left(r, 1, Q^{r-1}\right)$ is uniquely determined by i and Q^{r-1}, that H is random oracle, and thus that each $H(SIG_i\left(r, 1, Q^{r-1}\right)$ is a random 256-bit long string unique to each potential leader i of round r.

From this we can conclude that, if the 256-bit string Q^{r-1} were itself *randomly and independently selected*, than so would be the hashed credentials of all potential leaders of round r. In fact, all potential leaders are well defined, and so are their credentials (whether actually computed or not). Further, the set of potential leaders of round r is a random subset of the users of round $r-k$, and an honest potential leader i always properly constructs and propagates his message m^r_i, which contains i's credential. Thus, since the percentage of honest users is h, no matter what the malicious potential leaders might do (e.g., reveal or conceal their own credentials), the minimum hashed potential-leader credential belongs to a honest user, who is necessarily identified by everyone to be the leader ℓ^r of the round r. Accordingly, if the 256-bit string Q^{r-1} were itself *randomly and independently selected*, with probability exactly h (a) the leader ℓ^r is honest and (b) $\ell_j = \ell^r$ for all honest step-2 verifiers j.

In reality, the hashed credential are, yes, randomly selected, but depend on Q^{r-1}, which is

not randomly and independently selected. We shall prove in our analysis, however, that Q^{r-1} is sufficiently non-manipulatable to guarantee that the leader of a round is honest with probability h' sufficiently close to h: namely, $h' > h^2(1 + h - h^2)$. For instance, if $h = 80\%$, then $h' > .7424$.

Having identified the leader of the round (which they correctly do when the leader ℓ^r is honest), the task of the step-2 verifiers is to start executing the BA using as initial values what they believe to be the block of the leader. Actually, in order to minimize the amount of communication required, a verifier $j \in SV^{r,2}$ does not use, as his input value v'_j to the Byzantine protocol, the block B_j that he has actually received from ℓ_j (the user j believes to be the leader), but the the leader, but the hash of that block, that is, $v'_j = H(B_i)$. Thus, upon termination of the BA protocol, the verifiers of the last step do not compute the desired round-r block B^r, but compute (authenticate and propagate) $H(B^r)$. Accordingly, since $H(B^r)$ is digitally signed by sufficiently many verifiers of the last step of the BA protocol, the users in the system will realize that $H(B^r)$ is the hash of the new block. However, they must also retrieve (or wait for, since the execution is quite asynchronous) the block B^r itself, which the protocol ensures that is indeed available, no matter what the Adversary might do.

Asynchrony and Timing *Algorand'*$_1$ and *Algorand'*$_2$ have a significant degree of asynchrony. This is so because the Adversary has large latitude in scheduling the delivery of the messages being propagated. In addition, whether the total number of steps in a round is capped or not, there is the variance contribute by the number of steps actually taken.

As soon as he learns the certificates of $B^0, \ldots, B^{r-1}$, a user i computes Q^{r-1} and starts working on round r, checking whether he is a potential leader, or a verifier in some step s of round r.

Assuming that i must act at step s, in light of the discussed asynchrony, i relies on various strategies to ensure that he has sufficient information before he acts.

For instance, he might wait to receive at least a given number of messages from the verifiers of the previous step, or wait for a sufficient time to ensure that he receives the messages of sufficiently many verifiers of the previous step.

The Seed Q^r and the Look-Back Parameter k Recall that, ideally, the quantities Q^r should random and independent, although it will suffice for them to be sufficiently non-manipulatable by the Adversary.

At a first glance, we could choose Q^{r-1} to coincide with $H\left(PAY^{r-1}\right)$, and thus avoid to specify Q^{r-1} explicitly in B^{r-1}. An elementary analysis reveals, however, that malicious users may take advantage of this selection mechanism.[11] Some additional effort shows that myriads of other

[11] We are at the start of round $r-1$. Thus, $Q^{r-2} = PAY^{r-2}$ is publicly known, and the Adversary privately knows who are the potential leaders he controls. Assume that the Adversary controls 10% of the users, and that, with very high probability, a malicious user w is the potential leader of round $r-1$. That is, assume that $H\left(SIG_w\left(r-2, 1, Q^{r-2}\right)\right)$ is so small that it is highly improbable an honest potential leader will actually be the leader of round $r-1$. (Recall that, since we choose potential leaders via a secret cryptographic sortition mechanism, the Adversary does not know who the honest potential leaders are.) The Adversary, therefore, is in the enviable position of choosing the payset PAY' he wants, and have it become the official payset of round $r-1$. However, he can do more. He can also ensure that, with high probability, (*) one of his malicious users will be the leader also of round r, so that he can freely select what PAY^r will be. (And so on. At least for a long while, that is, as long as these high-probability events really occur.) To guarantee (*), the Adversary acts as follows. Let PAY' be the payset the Adversary prefers for round $r-1$. Then, he computes $H(PAY')$ and checks whether, for some already malicious player z, $SIG_z(r, 1, H(PAY'))$ is particularly small, that is, small enough that with very high probability z will be the leader of round r. If this is the case, then he instructs w to choose his candidate block to be

alternatives, based on traditional block quantities are easily exploitable by the Adversary to ensure that malicious leaders are very frequent. We instead specifically and inductively define our brand new quantity Q^r so as to be able to prove that it is non-manipulatable by the Adversary. Namely,

$$Q^r \triangleq H(SIG_{\ell^r}(Q^{r-1}), r), \text{ if } B^r \text{ is not the empty block, and } Q^r \triangleq H(Q^{r-1}, r) \text{ otherwise.}$$

The intuition of why this construction of Q^r works is as follows. Assume for a moment that Q^{r-1} is truly randomly and independently selected. Then, will so be Q^r? When ℓ^r is honest the answer is (roughly speaking) yes. This is so because

$$H(SIG_{\ell^r}(\cdot), r) : \{0,1\}^{256} \longrightarrow \{0,1\}^{256}$$

is a random function. When ℓ^r is malicious, however, Q^r is no longer univocally defined from Q^{r-1} and ℓ^r. There are at least two separate values for Q^r. One continues to be $Q^r \triangleq H(SIG_{\ell^r}(Q^{r-1}), r)$, and the other is $H(Q^{r-1}, r)$. Let us first argue that, while the second choice is somewhat arbitrary, a second choice is absolutely mandatory. The reason for this is that a malicious ℓ^r can always cause totally different candidate blocks to be received by the honest verifiers of the second step.[12] Once this is the case, it is easy to ensure that the block ultimately agreed upon via the BA protocol of round r will be the default one, and thus will not contain anyone's digital signature of Q^{r-1}. But the system must continue, and for this, it needs a leader for round r. If this leader is automatically and openly selected, then the Adversary will trivially corrupt him. If it is selected by the previous Q^{r-1} via the same process, than ℓ^r will again be the leader in round $r+1$. We specifically propose to use the same secret cryptographic sortition mechanism, but applied to a new Q-quantity: namely, $H(Q^{r-1}, r)$. By having this quantity to be the output of H guarantees that the output is random, and by including r as the second input of H, while all other uses of H have one or 3+ inputs, "guarantees" that such a Q^r is independently selected. Again, our specific choice of alternative Q^r does not matter, what matter is that ℓ^r has two choice for Q^r, and thus he can double his chances to have another malicious user as the next leader.

The options for Q^r may even be more numerous for the Adversary who controls a malicious ℓ^r. For instance, let x, y, and z be three malicious potential leaders of round r such that

$$H\left(\sigma_x^{r,1}\right) < H\left(\sigma_y^{r,1}\right) < H\left(\sigma_z^{r,1}\right)$$

and $H\left(\sigma_z^{r,1}\right)$ is particulary small. That is, so small that there is a good chance that $H\left(\sigma_z^{r,1}\right)$ is smaller of the hashed credential of every honest potential leader. Then, by asking x to hide his credential, the Adversary has a good chance of having y become the leader of round $r-1$. This implies that he has another option for Q^r: namely, $SIG_y\left(Q^{r-1}\right)$. Similarly, the Adversary may ask both x and y of withholding their credentials, so as to have z become the leader of round $r-1$ and gaining another option for Q^r: namely, $SIG_z\left(Q^{r-1}\right)$.

Of course, however, each of these and other options has a non-zero chance to fail, because the Adversary cannot predict the hash of the digital signatures of the honest potential users.

$B_i^{r-1} = (r-1, PAY', H(B^{r-2})$. Else, he has two other malicious users x and y to keep on generating a new payment $\wp'$, from one to the other, until, for some malicious user z (or even for some fixed user z) $H\left(SIG_z\left(PAY' \cup \{\wp\}\right)\right)$ is particularly small too. This experiment will stop quite quickly. And when it does the Adversary asks w to propose the candidate block $B_i^{r-1} = (r-1, PAY' \cup \{\wp\}, H(B^{r-2})$.

[12] For instance, to keep it simple (but extreme), "when the time of the second step is about to expire", ℓ^r could directly email a different candidate block B_i to each user i. This way, whoever the step-2 verifiers might be, they will have received totally different blocks.

A careful, Markov-chain-like analysis shows that, no matter what options the Adversary chooses to make at round $r-1$, *as long as he cannot inject new users in the system*, he cannot decrease the probability of an honest user to be the leader of round $r+40$ much below h. This is the reason for which we demand that the potential leaders of round r are users already existing in round $r-k$. It is a way to ensure that, at round $r-k$, the Adversary cannot alter by much the probability that an honest user become the leader of round r. In fact, no matter what users he may add to the system in rounds $r-k$ through r, they are ineligible to become potential leaders (and *a fortiori* the leader) of round r. Thus the look-back parameter k ultimately is a security parameter. (Although, as we shall see in section 7, it can also be a kind of "convenience parameter" as well.)

Ephemeral Keys Although the execution of our protocol cannot generate a fork, except with negligible probability, the Adversary could generate a fork, at the rth block, after the legitimate block r has been generated.

Roughly, once B^r has been generated, the Adversary has learned who the verifiers of each step of round r are. Thus, he could therefore corrupt all of them and oblige them to certify a new block $\widetilde{B^r}$. Since this fake block might be propagated only after the legitimate one, users that have been paying attention would not be fooled.[13] Nonetheless, $\widetilde{B^r}$ would be syntactically correct and we want to prevent from being manufactured.

We do so by means of a new rule. Essentially, the members of the verifier set $SV^{r,s}$ of a step s of round r use ephemeral public keys $pk_i^{r,s}$ to digitally sign their messages. These keys are single-use-only and their corresponding secret keys $sk_i^{r,s}$ are destroyed once used. This way, if a verifier is corrupted later on, the Adversary cannot force him to sign anything else he did not originally sign.

Naturally, we must ensure that it is impossible for the Adversary to compute a new key $\widetilde{pk_i^{r,s}}$ and convince an honest user that it is the right ephemeral key of verifier $i \in SV^{r,s}$ to use in step s.

4.2 Common Summary of Notations, Notions, and Parameters

Notations

- $r \geq 0$: the current round number.
- $s \geq 1$: the current step number in round r.
- B^r: the block generated in round r.
- PK^r: the set of public keys by the end of round $r-1$ and at the beginning of round r.
- S^r: the system status by the end of round $r-1$ and at the beginning of round r.[14]
- PAY^r: the payset contained in B^r.
- ℓ^r: round-r leader. ℓ^r chooses the payset PAY^r of round r (and determines the next Q^r).
- Q^r: the seed of round r, a quantity (i.e., binary string) that is generated at the end of round r and is used to choose verifiers for round $r+1$. Q^r is independent of the paysets in the blocks and cannot be manipulated by ℓ^r.

[13] Consider corrupting the news anchor of a major TV network, and producing and broadcasting today a newsreel showing secretary Clinton winning the last presidential election. Most of us would recognize it as a hoax. But someone getting out of a coma might be fooled.

[14] In a system that is not synchronous, the notion of "the end of round $r-1$" and "the beginning of round r" need to be carefully defined. Mathematically, PK^r and S^r are computed from the initial status S^0 and the blocks $B^1, \ldots, B^{r-1}$.

- $SV^{r,s}$: the set of verifiers chosen for step s of round r.
- SV^r: the set of verifiers chosen for round r, $SV^r = \cup_{s \geq 1} SV^{r,s}$.
- $MSV^{r,s}$ and $HSV^{r,s}$: respectively, the set of malicious verifiers and the set of honest verifiers in $SV^{r,s}$. $MSV^{r,s} \cup HSV^{r,s} = SV^{r,s}$ and $MSV^{r,s} \cap HSV^{r,s} = \emptyset$.
- $n_1 \in \mathbb{Z}^+$ and $n \in \mathbb{Z}^+$: respectively, the expected numbers of potential leaders in each $SV^{r,1}$, and the expected numbers of verifiers in each $SV^{r,s}$, for $s > 1$.
  Notice that $n_1 << n$, since we need at least one honest honest member in $SV^{r,1}$, but at least a majority of honest members in each $SV^{r,s}$ for $s > 1$.
- $h \in (0, 1)$: a constant greater than 2/3. h is the honesty ratio in the system. That is, the fraction of honest users or honest money, depending on the assumption used, in each PK^r is at least h.
- H: a cryptographic hash function, modelled as a random oracle.
- $\perp$: A special string of the same length as the output of H.
- $F \in (0, 1)$: the parameter specifying the allowed error probability. A probability $\leq F$ is considered "negligible", and a probability $\geq 1 - F$ is considered "overwhelming".
- $p_h \in (0, 1)$: the probability that the leader of a round r, ℓ^r, is honest. Ideally $p_h = h$. With the existence of the Adversary, the value of p_h will be determined in the analysis.
- $k \in \mathbb{Z}^+$: the look-back parameter. That is, round $r - k$ is where the verifiers for round r are chosen from —namely, $SV^r \subseteq PK^{r-k}$.[15]
- $p_1 \in (0, 1)$: for the first step of round r, a user in round $r - k$ is chosen to be in $SV^{r,1}$ with probability $p_1 \triangleq \frac{n_1}{|PK^{r-k}|}$.
- $p \in (0, 1)$: for each step $s > 1$ of round r, a user in round $r - k$ is chosen to be in $SV^{r,s}$ with probability $p \triangleq \frac{n}{|PK^{r-k}|}$.
- $CERT^r$: the certificate for B^r. It is a set of t_H signatures of $H(B^r)$ from proper verifiers in round r.
- $\overline{B^r} \triangleq (B^r, CERT^r)$ is a proven block.
  A user i *knows* B^r if he possesses (and successfully verifies) both parts of the proven block. Note that the $CERT^r$ seen by different users may be different.
- τ_i^r: the (local) time at which a user i knows B^r. In the Algorand protocol each user has his own clock. Different users' clocks need not be synchronized, but must have the same speed. Only for the purpose of the analysis, we consider a reference clock and measure the players' related times with respect to it.
- $\alpha_i^{r,s}$ and $\beta_i^{r,s}$: respectively the (local) time a user i starts and ends his execution of Step s of round r.
- Λ and λ: essentially, the upper-bounds to, respectively, the time needed to execute Step 1 and the time needed for any other step of the Algorand protocol.
  Parameter Λ upper-bounds the time to propagate a single 1MB block. (In our notation, $\Lambda = \lambda_{\rho,1MB}$. Recalling our notation, that we set $\rho = 1$ for simplicity, and that blocks are chosen to be at most 1MB-long, we have $\Lambda = \lambda_{1,1,1MB}$.)

[15] Strictly speaking, "$r - k$" should be "$\max\{0, r - k\}$".

Parameter λ upperbounds the time to propagate one small message per verifier in a Step $s > 1$. (Using, as in Bitcoin, elliptic curve signatures with 32B keys, a verifier message is 200B long. Thus, in our notation, $\lambda = \lambda_{n,\rho,200B}$.)
We assume that $\Lambda = O(\lambda)$.

Notions

- Verifier selection.

  For each round r and step $s > 1$, $SV^{r,s} \triangleq \{i \in PK^{r-k} : .H(SIG_i(r, s, Q^{r-1})) \leq p\}$. Each user $i \in PK^{r-k}$ privately computes his signature using his long-term key and decides whether $i \in SV^{r,s}$ or not. If $i \in SV^{r,s}$, then $SIG_i(r, s, Q^{r-1})$ is i's (r, s)-credential, compactly denoted by $\sigma_i^{r,s}$.

  For the first step of round r, $SV^{r,1}$ and $\sigma_i^{r,1}$ are similarly defined, with p replaced by p_1. The verifiers in $SV^{r,1}$ are *potential leaders*.

- Leader selection.

  User $i \in SV^{r,1}$ is the leader of round r, denoted by ℓ^r, if $H(\sigma_i^{r,1}) \leq H(\sigma_j^{r,1})$ for all potential leaders $j \in SV^{r,1}$. Whenever the hashes of two players' credentials are compared, in the unlikely event of ties, the protocol always breaks ties lexicographically according to the (long-term public keys of the) potential leaders.

  By definition, the hash value of player ℓ^r's credential is also the smallest among all users in PK^{r-k}. Note that a potential leader cannot privately decide whether he is the leader or not, without seeing the other potential leaders' credentials.

  Since the hash values are uniform at random, when $SV^{r,1}$ is non-empty, ℓ^r always exists and is honest with probability at least h. The parameter n_1 is large enough so as to ensure that each $SV^{r,1}$ is non-empty with overwhelming probability.

- Block structure.

  A non-empty block is of the form $B^r = (r, PAY^r, SIG_{\ell^r}(Q^{r-1}), H(B^{r-1}))$, and an empty block is of the form $B_\epsilon^r = (r, \emptyset, Q^{r-1}, H(B^{r-1}))$.

  Note that a non-empty block may still contain an empty payset PAY^r, if no payment occurs in this round or if the leader is malicious. However, a non-empty block implies that the identity of ℓ^r, his credential $\sigma_{\ell^r}^{r,1}$ and $SIG_{\ell^r}(Q^{r-1})$ have all been timely revealed. The protocol guarantees that, if the leader is honest, then the block will be non-empty with overwhelming probability.

- Seed Q^r.

  If B^r is non-empty, then $Q^r \triangleq H(SIG_{\ell^r}(Q^{r-1}), r)$, otherwise $Q^r \triangleq H(Q^{r-1}, r)$.

Parameters

- *Relationships among various parameters.*
  - The verifiers and potential leaders of round r are selected from the users in PK^{r-k}, where k is chosen so that the Adversary cannot predict Q^{r-1} back at round $r - k - 1$ with probability better than F: otherwise, he will be able to introduce malicious users for round $r - k$, all of which will be potential leaders/verifiers in round r, succeeding in

having a malicious leader or a malicious majority in $SV^{r,s}$ for some steps s desired by him.

— For Step 1 of each round r, n_1 is chosen so that with overwhelming probability, $SV^{r,1} \neq \emptyset$.

- *Example choices of important parameters.*
  — The outputs of H are 256-bit long.
  — $h = 80\%$, $n_1 = 35$.
  — $\Lambda = 1$ minute and $\lambda = 10$ seconds.
- *Initialization of the protocol.*
  The protocol starts at time 0 with $r = 0$. Since there does not exist "B^{-1}" or "$CERT^{-1}$", syntactically B^{-1} is a public parameter with its third component specifying Q^{-1}, and all users know B^{-1} at time 0.

5 *Algorand*$'_1$

In this section, we construct a version of *Algorand*$'$ working under the following assumption.

Honest Majority of Users Assumption: *More than 2/3 of the users in each PK^r are honest.*

In Section 8, we show how to replace the above assumption with the desired Honest Majority of Money assumption.

5.1 Additional Notations and Parameters

Notations

- $m \in \mathbb{Z}^+$: the maximum number of steps in the binary BA protocol, a multiple of 3.
- $L^r \leq m/3$: a random variable representing the number of Bernoulli trials needed to see a 1, when each trial is 1 with probability $\frac{p_h}{2}$ and there are at most $m/3$ trials. If all trials fail then $L^r \triangleq m/3$. L^r will be used to upper-bound the time needed to generate block B^r.
- $t_H = \frac{2n}{3} + 1$: the number of signatures needed in the ending conditions of the protocol.
- $CERT^r$: the certificate for B^r. It is a set of t_H signatures of $H(B^r)$ from proper verifiers in round r.

Parameters

- *Relationships among various parameters.*
  — For each step $s > 1$ of round r, n is chosen so that, with overwhelming probability,
  $$|HSV^{r,s}| > 2|MSV^{r,s}| \quad \text{and} \quad |HSV^{r,s}| + 4|MSV^{r,s}| < 2n.$$
  The closer to 1 the value of h is, the smaller n needs to be. In particular, we use (variants of) Chernoff bounds to ensure the desired conditions hold with overwhelming probability.
  — m is chosen such that $L^r < m/3$ with overwhelming probability.
- *Example choices of important parameters.*
  — $F = 10^{-12}$.
  — $n \approx 1500$, $k = 40$ and $m = 180$.

5.2 Implementing Ephemeral Keys in *Algorand*$'_1$

As already mentioned, we wish that a verifier $i \in SV^{r,s}$ digitally signs his message $m_i^{r,s}$ of step s in round r, relative to an ephemeral public key $pk_i^{r,s}$, using an ephemeral secrete key $sk_i^{r,s}$ that he promptly destroys after using. We thus need an efficient method to ensure that every user can verify that $pk_i^{r,s}$ is indeed the key to use to verify i's signature of $m_i^{r,s}$. We do so by a (to the best of our knowledge) new use of identity-based signature schemes.

At a high level, in such a scheme, a central authority A generates a public master key, PMK, and a corresponding secret master key, SMK. Given the identity, U, of a player U, A computes, via SMK, a secret signature key sk_U relative to the public key U, and privately gives sk_U to U. (Indeed, in an identity-based digital signature scheme, the public key of a user U is U itself!) This way, if A destroys SMK after computing the secret keys of the users he wants to enable to produce digital signatures, and does not keep any computed secret key, then U is the only one who can digitally sign messages relative to the public key U. Thus, anyone who knows "U's name", automatically knows U's public key, and thus can verify U's signatures (possibly using also the public master key PMK).

In our application, the authority A is user i, and the set of all possible users U coincides with the round-step pair (r, s) in —say— $S = \{i\} \times \{r', \ldots, r' + 10^6\} \times \{1, \ldots, m+3\}$, where r' is a given round, and $m + 3$ the upperbound to the number of steps that may occur within a round. This way, $pk_i^{r,s} \triangleq (i, r, s)$, so that everyone seeing i's signature $SIG^{r,s}_{pk_i^{r,s}}(m_i^{r,s})$ can, with overwhelming probability, immediately verify it for the first million rounds r following r'.

In other words, i first generates PMK and SMK. Then, he publicizes that PMK is i's master public key for any round $r \in [r', r' + 10^6]$, and uses SMK to privately produce and store the secret key $sk_i^{r,s}$ for each triple $(i, r, s) \in S$. This done, he destroys SMK. If he determines that he is not part of $SV^{r,s}$, then i may leave $sk_i^{r,s}$ alone (as the protocol does not require that he aunthenticates any message in Step s of round r). Else, i first uses $sk_i^{r,s}$ to digitally sign his message $m_i^{r,s}$, and then destroys $sk_i^{r,s}$.

Note that i can publicize his first public master key when he first enters the system. That is, the same payment $\wp$ that brings i into the system (at a round r' or at a round close to r'), may also specify, at i's request, that i's public master key for any round $r \in [r', r' + 10^6]$ is PMK —e.g., by including a pair of the form $(PMK, [r', r' + 10^6])$.

Also note that, since $m + 3$ is the maximum number of steps in a round, assuming that a round takes a minute, the stash of ephemeral keys so produced will last i for almost two years. At the same time, these ephemeral secret keys will not take i too long to produce. Using an elliptic-curve based system with 32B keys, each secret key is computed in a few microseconds. Thus, if $m + 3 = 180$, then all 180M secret keys can be computed in less than one hour.

When the current round is getting close to $r' + 10^6$, to handle the next million rounds, i generates a new (PMK', SMK') pair, and informs what his next stash of ephemeral keys is by —for example— having $SIG_i(PMK', [r' + 10^6 + 1, r' + 2 \cdot 10^6 + 1])$ enter a new block, either as a separate "transaction" or as some additional information that is part of a payment. By so doing, i informs everyone that he/she should use PMK' to verify i's ephemeral signatures in the next million rounds. And so on.

(Note that, following this basic approach, other ways for implementing ephemeral keys without using identity-based signatures are certainly possible. For instance, via Merkle trees.[16])

[16] In this method, i generates a public-secret key pair $(pk_i^{r,s}, sk_i^{r,s})$ for each round-step pair (r, s) in —say—

Other ways for implementing ephemeral keys are certainly possible —e.g., via Merkle trees.

5.3 Matching the Steps of *Algorand*$'_1$ with those of $BA^\star$

As we said, a round in *Algorand*$'_1$ has at most $m+3$ steps.

STEP 1. In this step, each potential leader i computes and propagates his candidate block B_i^r, together with his own credential, $\sigma_i^{r,1}$.

Recall that this credential *explicitly* identifies i. This is so, because $\sigma_i^{r,1} \triangleq SIG_i(r, 1, Q^{r-1})$.

Potential verifier i also propagates, as part of his message, his proper digital signature of $H(B_i^r)$. Not dealing with a payment or a credential, this signature of i is relative to his ephemeral public key $pk_i^{r,1}$: that is, he propagates $sig_{pk_i^{r,1}}(H(B_i^r))$.

Given our conventions, rather than propagating B_i^r and $sig_{pk_i^{r,1}}(H(B_i^r))$, he could have propagated $SIG_{pk_i^{r,1}}(H(B_i^r))$. However, in our analysis we need to have explicit access to $sig_{pk_i^{r,1}}(H(B_i^r))$.

STEPS 2. In this step, each verifier i sets ℓ_i^r to be the potential leader whose hashed credential is the smallest, and B_i^r to be the block proposed by ℓ_i^r. Since, for the sake of efficiency, we wish to agree on $H(B^r)$, rather than directly on B^r, i propagates the message he would have propagated in the first step of $BA^\star$ with initial value $v_i' = H(B_i^r)$. That is, he propagates v_i', after ephemerally signing it, of course. (Namely, after signing it relative to the right ephemeral public key, which in this case is $pk_i^{r,2}$.) Of course too, i also transmits his own credential.

Since the first step of $BA^\star$ consists of the first step of the graded consensus protocol GC, Step 2 of *Algorand*$'$ corresponds to the first step of GC.

STEPS 3. In this step, each verifier $i \in SV^{r,2}$ executes the second step of $BA^\star$. That is, he sends the same message he would have sent in the second step of GC. Again, i's message is ephemerally signed and accompanied by i's credential. (From now on, we shall omit saying that a verifier ephemerally signs his message and also propagates his credential.)

STEP 4. In this step, every verifier $i \in SV^{r,4}$ computes the output of GC, (v_i, g_i), and ephemerally signs and sends the same message he would have sent in the third step of $BA^\star$, that is, in the first step of $BBA^\star$, with initial bit 0 if $g_i = 2$, and 1 otherwise.

STEP $s = 5, \ldots, m+2$. Such a step, if ever reached, corresponds to step $s-1$ of $BA^\star$, and thus to step $s-3$ of $BBA^\star$.

Since our propagation model is sufficiently asynchronous, we must account for the possibility that, in the middle of such a step s, a verifier $i \in SV^{r,s}$ is reached by information proving him that block B^r has already been chosen. In this case, i stops his own execution of round r of *Algorand*$'$, and starts executing his round-$(r+1)$ instructions.

$\{r', \ldots, r' + 10^6\} \times \{1, \ldots, m+3\}$. Then he orders these public keys in a canonical way, stores the jth public key in the jth leaf of a Merkle tree, and computes the root value R_i, which he publicizes. When he wants to sign a message relative to key $pk_i^{r,s}$, i not only provides the actual signature, but also the authenticating path for $pk_i^{r,s}$ relative to R_i. Notice that this authenticating path also proves that $pk_i^{r,s}$ is stored in the jth leaf. The rest of the details can be easily filled.

Accordingly, the instructions of a verifier $i \in SV^{r,s}$, in addition to the instructions corresponding to Step $s-3$ of $BBA^\star$, include checking whether the execution of $BBA^\star$ has halted in a prior Step s'. Since $BBA^\star$ can only halt is a Coin-Fixed-to-0 Step or in a Coin-Fixed-to-1 step, the instructions distinguish whether

A (Ending Condition 0): $s' - 2 \equiv 0 \ mod \ 3$, or

B (Ending Condition 1): $s' - 2 \equiv 1 \ mod \ 3$.

In fact, in case A, the block B^r is non-empty, and thus additional instructions are necessary to ensure that i properly reconstructs B^r, together with its proper certificate $CERT^r$. In case B, the block B^r is empty, and thus i is instructed to set $B^r = B^r_\varepsilon = (r, \emptyset, H(Q^{r-1}, r), H(B^{r-1}))$, and to compute $CERT^r$.

If, during his execution of step s, i does not see any evidence that the block B^r has already been generated, then he sends the same message he would have sent in step $s-3$ of $BBA^\star$.

Step $m+3$. If, during step $m+3$, $i \in SV^{r,m+3}$ sees that the block B^r was already generated in a prior step s', then he proceeds just as explained above.

Else, rather then sending the same message he would have sent in step m of $BBA^\star$, i is instructed, based on the information in his possession, to compute B^r and its corresponding certificate $CERT^r$.

Recall, in fact, that we upperbound by $m+3$ the total number of steps of a round.

5.4 The Actual Protocol

Recall that, in each step s of a round r, a verifier $i \in SV^{r,s}$ uses his long-term public-secret key pair to produce his credential, $\sigma_i^{r,s} \triangleq SIG_i(r, s, Q^{r-1})$, as well as $SIG_i\left(Q^{r-1}\right)$ in case $s = 1$. Verifier i uses his ephemeral secret key $sk_i^{r,s}$ to sign his (r,s)-message $m_i^{r,s}$. For simplicity, when r and s are clear, we write $esig_i(x)$ rather than $sig_{pk_i^{r,s}}(x)$ to denote i's proper ephemeral signature of a value x in step s of round r, and write $ESIG_i(x)$ instead of $SIG_{pk_i^{r,s}}(x)$ to denote $(i, x, esig_i(x))$.

Step 1: Block Proposal

Instructions for every user $i \in PK^{r-k}$: User i starts his own Step 1 of round r as soon as he knows B^{r-1}.

- User i computes Q^{r-1} from the third component of B^{r-1} and checks whether $i \in SV^{r,1}$ or not.
- If $i \notin SV^{r,1}$, then i stops his own execution of Step 1 right away.
- If $i \in SV^{r,1}$, that is, if i is a potential leader, then he collects the round-r payments that have been propagated to him so far and computes a maximal payset PAY_i^r from them. Next, he computes his "candidate block" $B_i^r = (r, PAY_i^r, SIG_i(Q^{r-1}), H(B^{r-1}))$. Finally, he computes the message $m_i^{r,1} = (B_i^r, esig_i(H(B_i^r)), \sigma_i^{r,1})$, destroys his ephemeral secret key $sk_i^{r,1}$, and then propagates $m_i^{r,1}$.

Remark. In practice, to shorten the global execution of Step 1, it is important that the $(r,1)$-messages are *selectively propagated*. That is, for every user i in the system, for the first $(r,1)$-message that he ever receives and successfully verifies,[17] player i propagates it as usual. For all the other $(r,1)$-messages that player i receives and successfully verifies, he propagates it only if the hash value of the credential it contains is the *smallest* among the hash values of the credentials contained in all $(r,1)$-messages he has received and successfully verified so far. Furthermore, as suggested by Georgios Vlachos, it is useful that each potential leader i also propagates his credential $\sigma_i^{r,1}$ separately: those small messages travel faster than blocks, ensure timely propagation of the $m_j^{r,1}$'s where the contained credentials have small hash values, while make those with large hash values disappear quickly.

Step 2: The First Step of the Graded Consensus Protocol GC

Instructions for every user $i \in PK^{r-k}$: User i starts his own Step 2 of round r as soon as he knows B^{r-1}.

- User i computes Q^{r-1} from the third component of B^{r-1} and checks whether $i \in SV^{r,2}$ or not.
- If $i \notin SV^{r,2}$ then i stops his own execution of Step 2 right away.
- If $i \in SV^{r,2}$, then after waiting an amount of time $t_2 \triangleq \lambda + \Lambda$, i acts as follows.
  1. He finds the user ℓ such that $H(\sigma_\ell^{r,1}) \leq H(\sigma_j^{r,1})$ for all credentials $\sigma_j^{r,1}$ that are part of the successfully verified $(r,1)$-messages he has received so far.[a]
  2. If he has received from ℓ a valid message $m_\ell^{r,1} = (B_\ell^r, esig_\ell(H(B_\ell^r)), \sigma_\ell^{r,1})$,[b] then i sets $v_i' \triangleq H(B_\ell^r)$; otherwise i sets $v_i' \triangleq \perp$.
  3. i computes the message $m_i^{r,2} \triangleq (ESIG_i(v_i'), \sigma_i^{r,2})$,[c] destroys his ephemeral secret key $sk_i^{r,2}$, and then propagates $m_i^{r,2}$.

[a] Essentially, user i privately decides that the leader of round r is user ℓ.

[b] Again, player ℓ's signatures and the hashes are all successfully verified, and PAY_ℓ^r in B_ℓ^r is a valid payset for round r —although i does not check whether PAY_ℓ^r is maximal for ℓ or not.

[c] The message $m_i^{r,2}$ signals that player i considers v_i' to be the hash of the next block, or considers the next block to be empty.

[17] That is, all the signatures are correct and both the block and its hash are valid —although i does not check whether the included payset is maximal for its proposer or not.

Step 3: The Second Step of GC

Instructions for every user $i \in PK^{r-k}$: User i starts his own Step 3 of round r as soon as he knows B^{r-1}.

- User i computes Q^{r-1} from the third component of B^{r-1} and checks whether $i \in SV^{r,3}$ or not.
- If $i \notin SV^{r,3}$, then i stops his own execution of Step 3 right away.
- If $i \in SV^{r,3}$, then after waiting an amount of time $t_3 \triangleq t_2 + 2\lambda = 3\lambda + \Lambda$, i acts as follows.
  1. If there exists a value $v' \neq \perp$ such that, among all the valid messages $m_j^{r,2}$ he has received, more than 2/3 of them are of the form $(ESIG_j(v'), \sigma_j^{r,2})$, without any contradiction,[a] then he computes the message $m_i^{r,3} \triangleq (ESIG_i(v'), \sigma_i^{r,3})$. Otherwise, he computes $m_i^{r,3} \triangleq (ESIG_i(\perp), \sigma_i^{r,3})$.
  2. i destroys his ephemeral secret key $sk_i^{r,3}$, and then propagates $m_i^{r,3}$.

[a] That is, he has not received two valid messages containing $ESIG_j(v')$ and a different $ESIG_j(v'')$ respectively, from a player j. Here and from here on, except in the Ending Conditions defined later, whenever an honest player wants messages of a given form, messages contradicting each other are never counted or considered valid.

Step 4: Output of GC and The First Step of $BBA^\star$

Instructions for every user $i \in PK^{r-k}$: User i starts his own Step 4 of round r as soon as he knows B^{r-1}.

- User i computes Q^{r-1} from the third component of B^{r-1} and checks whether $i \in SV^{r,4}$ or not.
- If $i \notin SV^{r,4}$, then i his stops his own execution of Step 4 right away.
- If $i \in SV^{r,4}$, then after waiting an amount of time $t_4 \triangleq t_3 + 2\lambda = 5\lambda + \Lambda$, i acts as follows.
  1. He computes v_i and g_i, the output of GC, as follows.
     (a) If there exists a value $v' \neq \perp$ such that, among all the valid messages $m_j^{r,3}$ he has received, more than 2/3 of them are of the form $(ESIG_j(v'), \sigma_j^{r,3})$, then he sets $v_i \triangleq v'$ and $g_i \triangleq 2$.
     (b) Otherwise, if there exists a value $v' \neq \perp$ such that, among all the valid messages $m_j^{r,3}$ he has received, more than 1/3 of them are of the form $(ESIG_j(v'), \sigma_j^{r,3})$, then he sets $v_i \triangleq v'$ and $g_i \triangleq 1$.[a]
     (c) Else, he sets $v_i \triangleq H(B_\epsilon^r)$ and $g_i \triangleq 0$.
  2. He computes b_i, the input of $BBA^\star$, as follows:
     $b_i \triangleq 0$ if $g_i = 2$, and $b_i \triangleq 1$ otherwise.
  3. He computes the message $m_i^{r,4} \triangleq (ESIG_i(b_i), ESIG_i(v_i), \sigma_i^{r,4})$, destroys his ephemeral secret key $sk_i^{r,4}$, and then propagates $m_i^{r,4}$.

[a] It can be proved that the v' in case (b), if exists, must be unique.

Step s, $5 \leq s \leq m+2$, $s-2 \equiv 0 \mod 3$: A Coin-Fixed-To-0 Step of $BBA^\star$

Instructions for every user $i \in PK^{r-k}$: User i starts his own Step s of round r as soon as he knows B^{r-1}.

- User i computes Q^{r-1} from the third component of B^{r-1} and checks whether $i \in SV^{r,s}$.
- If $i \notin SV^{r,s}$, then i stops his own execution of Step s right away.
- If $i \in SV^{r,s}$ then he acts as follows.
  - He waits until an amount of time $t_s \triangleq t_{s-1} + 2\lambda = (2s-3)\lambda + \Lambda$ has passed.
  - *Ending Condition 0:* If, during such waiting and at any point of time, there exists a string $v \neq \perp$ and a step s' such that
    (a) $5 \leq s' \leq s$, $s'-2 \equiv 0 \mod 3$ —that is, Step s' is a Coin-Fixed-To-0 step,
    (b) i has received at least $t_H = \frac{2n}{3} + 1$ valid messages $m_j^{r,s'-1} = (ESIG_j(0), ESIG_j(v), \sigma_j^{r,s'-1})$,[a] and
    (c) i has received a valid message $m_j^{r,1} = (B_j^r, esig_j(H(B_j^r)), \sigma_j^{r,1})$ with $v = H(B_j^r)$,
    then, i stops his own execution of Step s (and in fact of round r) right away without propagating anything; sets $B^r = B_j^r$; and sets his own $CERT^r$ to be the set of messages $m_j^{r,s'-1}$ of sub-step (b).[b]
  - *Ending Condition 1:* If, during such waiting and at any point of time, there exists a step s' such that
    (a') $6 \leq s' \leq s$, $s'-2 \equiv 1 \mod 3$ —that is, Step s' is a Coin-Fixed-To-1 step, and
    (b') i has received at least t_H valid messages $m_j^{r,s'-1} = (ESIG_j(1), ESIG_j(v_j), \sigma_j^{r,s'-1})$,[c]
    then, i stops his own execution of Step s (and in fact of round r) right away without propagating anything; sets $B^r = B_\epsilon^r$; and sets his own $CERT^r$ to be the set of messages $m_j^{r,s'-1}$ of sub-step (b').
  - Otherwise, at the end of the wait, user i does the following.
    He sets v_i to be the majority vote of the v_j's in the second components of all the valid $m_j^{r,s-1}$'s he has received.
    He computes b_i as follows.
    If more than 2/3 of all the valid $m_j^{r,s-1}$'s he has received are of the form $(ESIG_j(0), ESIG_j(v_j), \sigma_j^{r,s-1})$, then he sets $b_i \triangleq 0$.
    Else, if more than 2/3 of all the valid $m_j^{r,s-1}$'s he has received are of the form $(ESIG_j(1), ESIG_j(v_j), \sigma_j^{r,s-1})$, then he sets $b_i \triangleq 1$.
    Else, he sets $b_i \triangleq 0$.
    He computes the message $m_i^{r,s} \triangleq (ESIG_i(b_i), ESIG_i(v_i), \sigma_i^{r,s})$, destroys his ephemeral secret key $sk_i^{r,s}$, and then propagates $m_i^{r,s}$.

[a] Such a message from player j is counted even if player i has also received a message from j signing for 1. Similar things for Ending Condition 1. As shown in the analysis, this is done to ensure that all honest users know B^r within time λ from each other.

[b] User i now knows B^r and his own round r finishes. He still helps propagating messages as a generic user, but does not initiate any propagation as a (r,s)-verifier. In particular, he has helped propagating all messages in his $CERT^r$, which is enough for our protocol. Note that he should also set $b_i \triangleq 0$ for the binary BA protocol, but b_i is not needed in this case anyway. Similar things for all future instructions.

[c] In this case, it does not matter what the v_j's are.

Step s, $6 \leq s \leq m+2$, $s-2 \equiv 1 \mod 3$: A Coin-Fixed-To-1 Step of $BBA^\star$

Instructions for every user $i \in PK^{r-k}$: User i starts his own Step s of round r as soon as he knows B^{r-1}.

- User i computes Q^{r-1} from the third component of B^{r-1} and checks whether $i \in SV^{r,s}$ or not.
- If $i \notin SV^{r,s}$, then i stops his own execution of Step s right away.
- If $i \in SV^{r,s}$ then he does the follows.
  - He waits until an amount of time $t_s \triangleq (2s-3)\lambda + \Lambda$ has passed.
  - *Ending Condition 0:* The same instructions as Coin-Fixed-To-0 steps.
  - *Ending Condition 1:* The same instructions as Coin-Fixed-To-0 steps.
  - Otherwise, at the end of the wait, user i does the following.

    He sets v_i to be the majority vote of the v_j's in the second components of all the valid $m_j^{r,s-1}$'s he has received.

    He computes b_i as follows.

    If more than 2/3 of all the valid $m_j^{r,s-1}$'s he has received are of the form $(ESIG_j(0), ESIG_j(v_j), \sigma_j^{r,s-1})$, then he sets $b_i \triangleq 0$.

    Else, if more than 2/3 of all the valid $m_j^{r,s-1}$'s he has received are of the form $(ESIG_j(1), ESIG_j(v_j), \sigma_j^{r,s-1})$, then he sets $b_i \triangleq 1$.

    Else, he sets $b_i \triangleq 1$.

    He computes the message $m_i^{r,s} \triangleq (ESIG_i(b_i), ESIG_i(v_i), \sigma_i^{r,s})$, destroys his ephemeral secret key $sk_i^{r,s}$, and then propagates $m_i^{r,s}$.

Step s, $7 \leq s \leq m+2$, $s-2 \equiv 2 \mod 3$: A Coin-Genuinely-Flipped Step of $BBA^\star$

Instructions for every user $i \in PK^{r-k}$: User i starts his own Step s of round r as soon as he knows B^{r-1}.

- User i computes Q^{r-1} from the third component of B^{r-1} and checks whether $i \in SV^{r,s}$ or not.
- If $i \notin SV^{r,s}$, then i stops his own execution of Step s right away.
- If $i \in SV^{r,s}$ then he does the follows.
  - He waits until an amount of time $t_s \triangleq (2s-3)\lambda + \Lambda$ has passed.
  - *Ending Condition 0:* The same instructions as Coin-Fixed-To-0 steps.
  - *Ending Condition 1:* The same instructions as Coin-Fixed-To-0 steps.
  - Otherwise, at the end of the wait, user i does the following.

    He sets v_i to be the majority vote of the v_j's in the second components of all the valid $m_j^{r,s-1}$'s he has received.

    He computes b_i as follows.

    If more than 2/3 of all the valid $m_j^{r,s-1}$'s he has received are of the form $(ESIG_j(0), ESIG_j(v_j), \sigma_j^{r,s-1})$, then he sets $b_i \triangleq 0$.

    Else, if more than 2/3 of all the valid $m_j^{r,s-1}$'s he has received are of the form $(ESIG_j(1), ESIG_j(v_j), \sigma_j^{r,s-1})$, then he sets $b_i \triangleq 1$.

    Else, let $SV_i^{r,s-1}$ be the set of $(r, s-1)$-verifiers from whom he has received a valid message $m_j^{r,s-1}$. He sets $b_i \triangleq \mathtt{lsb}(\min_{j \in SV_i^{r,s-1}} H(\sigma_j^{r,s-1}))$.

    He computes the message $m_i^{r,s} \triangleq (ESIG_i(b_i), ESIG_i(v_i), \sigma_i^{r,s})$, destroys his ephemeral secret key $sk_i^{r,s}$, and then propagates $m_i^{r,s}$.

Step $m+3$: The Last Step of $BBA^\star$ [a]

Instructions for every user $i \in PK^{r-k}$: User i starts his own Step $m+3$ of round r as soon as he knows B^{r-1}.

- User i computes Q^{r-1} from the third component of B^{r-1} and checks whether $i \in SV^{r,m+3}$ or not.
- If $i \notin SV^{r,m+3}$, then i stops his own execution of Step $m+3$ right away.
- If $i \in SV^{r,m+3}$ then he does the follows.
  - He waits until an amount of time $t_{m+3} \triangleq t_{m+2} + 2\lambda = (2m+3)\lambda + \Lambda$ has passed.
  - *Ending Condition 0:* The same instructions as Coin-Fixed-To-0 steps.
  - *Ending Condition 1:* The same instructions as Coin-Fixed-To-0 steps.
  - Otherwise, at the end of the wait, user i does the following.
    He sets $out_i \triangleq 1$ and $B^r \triangleq B^r_\epsilon$.
    He computes the message $m_i^{r,m+3} = (ESIG_i(out_i), ESIG_i(H(B^r)), \sigma_i^{r,m+3})$, destroys his ephemeral secret key $sk_i^{r,m+3}$, and then propagates $m_i^{r,m+3}$ to certify B^r.[b]

[a] With overwhelming probability $BBA^\star$ has ended before this step, and we specify this step for completeness.

[b] A certificate from Step $m+3$ does not have to include $ESIG_i(out_i)$. We include it for uniformity only: the certificates now have a uniform format no matter in which step they are generated.

Reconstruction of the Round-r Block by Non-Verifiers

Instructions for every user i in the system: User i starts his own round r as soon as he knows B^{r-1}, and waits for block information as follows.

- If, during such waiting and at any point of time, there exists a string v and a step s' such that

  (a) $5 \leq s' \leq m+3$ with $s'-2 \equiv 0 \mod 3$,

  (b) i has received at least t_H valid messages $m_j^{r,s'-1} = (ESIG_j(0), ESIG_j(v), \sigma_j^{r,s'-1})$, and

  (c) i has received a valid message $m_j^{r,1} = (B_j^r, esig_j(H(B_j^r)), \sigma_j^{r,1})$ with $v = H(B_j^r)$,

  then, i stops his own execution of round r right away; sets $B^r = B_j^r$; and sets his own $CERT^r$ to be the set of messages $m_j^{r,s'-1}$ of sub-step (b).

- If, during such waiting and at any point of time, there exists a step s' such that

  (a') $6 \leq s' \leq m+3$ with $s'-2 \equiv 1 \mod 3$, and

  (b') i has received at least t_H valid messages $m_j^{r,s'-1} = (ESIG_j(1), ESIG_j(v_j), \sigma_j^{r,s'-1})$,

  then, i stops his own execution of round r right away; sets $B^r = B_\epsilon^r$; and sets his own $CERT^r$ to be the set of messages $m_j^{r,s'-1}$ of sub-step (b').

- If, during such waiting and at any point of time, i has received at least t_H valid messages $m_j^{r,m+3} = (ESIG_j(1), ESIG_j(H(B_\epsilon^r)), \sigma_j^{r,m+3})$, then i stops his own execution of round r right away, sets $B^r = B_\epsilon^r$, and sets his own $CERT^r$ to be the set of messages $m_j^{r,m+3}$ for 1 and $H(B_\epsilon^r)$.

5.5 Analysis of *Algorand'*$_1$

We introduce the following notations for each round $r \geq 0$, used in the analysis.

- Let T^r be the time when the first honest user knows B^{r-1}.
- Let I^{r+1} be the interval $[T^{r+1}, T^{r+1} + \lambda]$.

Note that $T^0 = 0$ by the initialization of the protocol. For each $s \geq 1$ and $i \in SV^{r,s}$, recall that $\alpha_i^{r,s}$ and $\beta_i^{r,s}$ are respectively the starting time and the ending time of player i's step s. Moreover, recall that $t_s = (2s-3)\lambda + \Lambda$ for each $2 \leq s \leq m+3$. In addition, let $I^0 \triangleq \{0\}$ and $t_1 \triangleq 0$.

Finally, recall that $L^r \leq m/3$ is a random variable representing the number of Bernoulli trials needed to see a 1, when each trial is 1 with probability $\frac{p_h}{2}$ and there are at most $m/3$ trials. If all trials fail then $L^r \triangleq m/3$.

In the analysis we ignore computation time, as it is in fact negligible relative to the time needed to propagate messages. In any case, by using slightly larger λ and Λ, the computation time can be incorporated into the analysis directly. Most of the statements below hold "with overwhelming probability," and we may not repeatedly emphasize this fact in the analysis.

5.6 Main Theorem

Theorem 5.1. *The following properties hold with overwhelming probability for each round $r \geq 0$:*

1. *All honest users agree on the same block B^r.*
2. *When the leader ℓ^r is honest, the block B^r is generated by ℓ^r, B^r contains a maximal payset received by ℓ^r by time $\alpha_{\ell^r}^{r,1}$, $T^{r+1} \leq T^r + 8\lambda + \Lambda$ and all honest users know B^r in the time interval I^{r+1}.*
3. *When the leader ℓ^r is malicious, $T^{r+1} \leq T^r + (6L^r + 10)\lambda + \Lambda$ and all honest users know B^r in the time interval I^{r+1}.*
4. *$p_h = h^2(1 + h - h^2)$ for L^r, and the leader ℓ^r is honest with probability at least p_h.*

Before proving our main theorem, let us make two remarks.

Remarks.

- *Block-Generation and True Latency.* The time to generate block B^r is defined to be $T^{r+1} - T^r$. That is, it is defined to be the difference between the first time some honest user learns B^r and the first time some honest user learns B^{r-1}. When the round-r leader is honest, Property 2 our main theorem guarantees that the *exact* time to generate B^r is $8\lambda + \Lambda$ time, no matter what the precise value of $h > 2/3$ may be. When the leader is malicious, Property 3 implies that the *expected* time to generate B^r is upperbounded by $(\frac{12}{p_h} + 10)\lambda + \Lambda$, again no matter the precise value of h.[18] However, the expected time to generate B^r depends on the precise value of h. Indeed, by Property 4, $p_h = h^2(1 + h - h^2)$ and the leader is honest with probability at least p_h, thus

  $$\mathbb{E}[T^{r+1} - T^r] \leq h^2(1 + h - h^2) \cdot (8\lambda + \Lambda) + (1 - h^2(1 + h - h^2))((\frac{12}{h^2(1 + h - h^2)} + 10)\lambda + \Lambda).$$

  For instance, if $h = 80\%$, then $\mathbb{E}[T^{r+1} - T^r] \leq 12.7\lambda + \Lambda$.

- *λ vs. Λ.* Note that the size of the messages sent by the verifiers in a step *Algorand′* is dominated by the length of the digital signature keys, which can remain fixed, even when the number of users is enormous. Also note that, in any step $s > 1$, the same expected number n of verifiers can be used whether the number of users is 100K, 100M, or 100M. This is so because n solely depends on h and F. In sum, therefore, barring a sudden need to increase secret key length, the value of λ should remain the same no matter how large the number of users may be in the foreseeable future.

  By contrast, for any transaction rate, the number of transactions grows with the number of users. Therefore, to process all new transactions in a timely fashion, the size of a block should also grow with the number of users, causing Λ to grow too. Thus, in the long run, we should have $\lambda << \Lambda$. Accordingly, it is proper to have a larger coefficient for λ, and actually a coefficient of 1 for Λ.

Proof of Theorem 5.1. We prove Properties 1–3 by induction: assuming they hold for round $r - 1$ (without loss of generality, they automatically hold for "round -1" when $r = 0$), we prove them for round r.

[18] Indeed, $\mathbb{E}[T^{r+1} - T^r] \leq (6\mathbb{E}[L^r] + 10)\lambda + \Lambda = (6 \cdot \frac{2}{p_h} + 10)\lambda + \Lambda = (\frac{12}{p_h} + 10)\lambda + \Lambda$.

Since B^{r-1} is uniquely defined by the inductive hypothesis, the set $SV^{r,s}$ is uniquely defined for each step s of round r. By the choice of n_1, $SV^{r,1} \neq \emptyset$ with overwhelming probability. We now state the following two lemmas, proved in Sections 5.7 and 5.8. Throughout the induction and in the proofs of the two lemmas, the analysis for round 0 is almost the same as the inductive step, and we will highlight the differences when they occur.

Lemma 5.2. *[Completeness Lemma] Assuming Properties 1–3 hold for round $r-1$, when the leader ℓ^r is honest, with overwhelming probability,*

- *All honest users agree on the same block B^r, which is generated by ℓ^r and contains a maximal payset received by ℓ^r by time $\alpha_{\ell^r}^{r,1} \in I^r$; and*
- *$T^{r+1} \leq T^r + 8\lambda + \Lambda$ and all honest users know B^r in the time interval I^{r+1}.*

Lemma 5.3. *[Soundness Lemma] Assuming Properties 1–3 hold for round $r-1$, when the leader ℓ^r is malicious, with overwhelming probability, all honest users agree on the same block B^r, $T^{r+1} \leq T^r + (6L^r + 10)\lambda + \Lambda$ and all honest users know B^r in the time interval I^{r+1}.*

Properties 1–3 hold by applying Lemmas 5.2 and 5.3 to $r = 0$ and to the inductive step. Finally, we restate Property 4 as the following lemma, proved in Section 5.9.

Lemma 5.4. *Given Properties 1–3 for each round before r, $p_h = h^2(1 + h - h^2)$ for L^r, and the leader ℓ^r is honest with probability at least p_h.*

Combining the above three lemmas together, Theorem 5.1 holds. ■

The lemma below states several important properties about round r given the inductive hypothesis, and will be used in the proofs of the above three lemmas.

Lemma 5.5. *Assume Properties 1–3 hold for round $r-1$. For each step $s \geq 1$ of round r and each honest verifier $i \in HSV^{r,s}$, we have that*

(a) $\alpha_i^{r,s} \in I^r$;

(b) if player i has waited an amount of time t_s, then $\beta_i^{r,s} \in [T^r + t_s, T^r + \lambda + t_s]$ for $r > 0$ and $\beta_i^{r,s} = t_s$ for $r = 0$; and

(c) if player i has waited an amount of time t_s, then by time $\beta_i^{r,s}$, he has received all messages sent by all honest verifiers $j \in HSV^{r,s'}$ for all steps $s' < s$.

Moreover, for each step $s \geq 3$, we have that

(d) there do not exist two different players $i, i' \in SV^{r,s}$ and two different values v, v' of the same length, such that both players have waited an amount of time t_s, more than 2/3 of all the valid messages $m_j^{r,s-1}$ player i receives have signed for v, and more than 2/3 of all the valid messages $m_j^{r,s-1}$ player i' receives have signed for v'.

Proof. Property (a) follows directly from the inductive hypothesis, as player i knows B^{r-1} in the time interval I^r and starts his own step s right away. Property (b) follows directly from (a): since player i has waited an amount of time t_s before acting, $\beta_i^{r,s} = \alpha_i^{r,s} + t_s$. Note that $\alpha_i^{r,s} = 0$ for $r = 0$.

We now prove Property (c). If $s = 2$, then by Property (b), for all verifiers $j \in HSV^{r,1}$ we have

$$\beta_i^{r,s} = \alpha_i^{r,s} + t_s \geq T^r + t_s = T^r + \lambda + \Lambda \geq \beta_j^{r,1} + \Lambda.$$

Since each verifier $j \in HSV^{r,1}$ sends his message at time $\beta_j^{r,1}$ and the message reaches all honest users in at most Λ time, by time $\beta_i^{r,s}$ player i has received the messages sent by all verifiers in $HSV^{r,1}$ as desired.

If $s > 2$, then $t_s = t_{s-1} + 2\lambda$. By Property (b), for all steps $s' < s$ and all verifiers $j \in HSV^{r,s'}$,

$$\beta_i^{r,s} = \alpha_i^{r,s} + t_s \geq T^r + t_s = T^r + t_{s-1} + 2\lambda \geq T^r + t_{s'} + 2\lambda = T^r + \lambda + t_{s'} + \lambda \geq \beta_j^{r,s'} + \lambda.$$

Since each verifier $j \in HSV^{r,s'}$ sends his message at time $\beta_j^{r,s'}$ and the message reaches all honest users in at most λ time, by time $\beta_i^{r,s}$ player i has received all messages sent by all honest verifiers in $HSV^{r,s'}$ for all $s' < s$. Thus Property (c) holds.

Finally, we prove Property (d). Note that the verifiers $j \in SV^{r,s-1}$ sign at most two things in Step $s-1$ using their ephemeral secret keys: a value v_j of the same length as the output of the hash function, and also a bit $b_j \in \{0,1\}$ if $s - 1 \geq 4$. That is why in the statement of the lemma we require that v and v' have the same length: many verifiers may have signed both a hash value v and a bit b, thus both pass the 2/3 threshold.

Assume for the sake of contradiction that there exist the desired verifiers i, i' and values v, v'. Note that some malicious verifiers in $MSV^{r,s-1}$ may have signed both v and v', but each honest verifier in $HSV^{r,s-1}$ has signed at most one of them. By Property (c), both i and i' have received all messages sent by all honest verifiers in $HSV^{r,s-1}$.

Let $HSV^{r,s-1}(v)$ be the set of honest $(r, s-1)$-verifiers who have signed v, $MSV_i^{r,s-1}$ the set of malicious $(r, s-1)$-verifiers from whom i has received a valid message, and $MSV_i^{r,s-1}(v)$ the subset of $MSV_i^{r,s-1}$ from whom i has received a valid message signing v. By the requirements for i and v, we have

$$ratio \triangleq \frac{|HSV^{r,s-1}(v)| + |MSV_i^{r,s-1}(v)|}{|HSV^{r,s-1}| + |MSV_i^{r,s-1}|} > \frac{2}{3}. \tag{1}$$

We first show

$$|MSV_i^{r,s-1}(v)| \leq |HSV^{r,s-1}(v)|. \tag{2}$$

Assuming otherwise, by the relationships among the parameters, with overwhelming probability $|HSV^{r,s-1}| > 2|MSV^{r,s-1}| \geq 2|MSV_i^{r,s-1}|$, thus

$$ratio < \frac{|HSV^{r,s-1}(v)| + |MSV_i^{r,s-1}(v)|}{3|MSV_i^{r,s-1}|} < \frac{2|MSV_i^{r,s-1}(v)|}{3|MSV_i^{r,s-1}|} \leq \frac{2}{3},$$

contradicting Inequality 1.

Next, by Inequality 1 we have

$$\begin{aligned} & 2|HSV^{r,s-1}| + 2|MSV_i^{r,s-1}| < 3|HSV^{r,s-1}(v)| + 3|MSV_i^{r,s-1}(v)| \\ \leq\ & 3|HSV^{r,s-1}(v)| + 2|MSV_i^{r,s-1}| + |MSV_i^{r,s-1}(v)|. \end{aligned}$$

Combining with Inequality 2,

$$2|HSV^{r,s-1}| < 3|HSV^{r,s-1}(v)| + |MSV_i^{r,s-1}(v)| \leq 4|HSV^{r,s-1}(v)|,$$

which implies

$$|HSV^{r,s-1}(v)| > \frac{1}{2}|HSV^{r,s-1}|.$$

Similarly, by the requirements for i' and v', we have

$$|HSV^{r,s-1}(v')| > \frac{1}{2}|HSV^{r,s-1}|.$$

Since an honest verifier $j \in HSV^{r,s-1}$ destroys his ephemeral secret key $sk_j^{r,s-1}$ before propagating his message, the Adversary cannot forge j's signature for a value that j did not sign, after learning that j is a verifier. Thus, the two inequalities above imply $|HSV^{r,s-1}| \geq |HSV^{r,s-1}(v)| + |HSV^{r,s-1}(v')| > |HSV^{r,s-1}|$, a contradiction. Accordingly, the desired i, i', v, v' do not exist, and Property (d) holds. ■

5.7 The Completeness Lemma

Lemma 5.2. [Completeness Lemma, restated] *Assuming Properties 1–3 hold for round $r-1$, when the leader ℓ^r is honest, with overwhelming probability,*

- *All honest users agree on the same block B^r, which is generated by ℓ^r and contains a maximal payset received by ℓ^r by time $\alpha_{\ell^r}^{r,1} \in I^r$; and*
- *$T^{r+1} \leq T^r + 8\lambda + \Lambda$ and all honest users know B^r in the time interval I^{r+1}.*

Proof. By the inductive hypothesis and Lemma 5.5, for each step s and verifier $i \in HSV^{r,s}$, $\alpha_i^{r,s} \in I^r$. Below we analyze the protocol step by step.

Step 1. By definition, every honest verifier $i \in HSV^{r,1}$ propagates the desired message $m_i^{r,1}$ at time $\beta_i^{r,1} = \alpha_i^{r,1}$, where $m_i^{r,1} = (B_i^r, esig_i(H(B_i^r)), \sigma_i^{r,1})$, $B_i^r = (r, PAY_i^r, SIG_i(Q^{r-1}), H(B^{r-1}))$, and PAY_i^r is a maximal payset among all payments that i has seen by time $\alpha_i^{r,1}$.

Step 2. Arbitrarily fix an honest verifier $i \in HSV^{r,2}$. By Lemma 5.5, when player i is done waiting at time $\beta_i^{r,2} = \alpha_i^{r,2} + t_2$, he has received all messages sent by verifiers in $HSV^{r,1}$, including $m_{\ell^r}^{r,1}$. By the definition of ℓ^r, there does not exist another player in PK^{r-k} whose credential's hash value is smaller than $H(\sigma_{\ell^r}^{r,1})$. Of course, the Adversary can corrupt ℓ^r after seeing that $H(\sigma_{\ell^r}^{r,1})$ is very small, but by that time player ℓ^r has destroyed his ephemeral key and the message $m_{\ell^r}^{r,1}$ has been propagated. Thus verifier i sets his own leader to be player ℓ^r. Accordingly, at time $\beta_i^{r,2}$, verifier i propagates $m_i^{r,2} = (ESIG_i(v_i'), \sigma_i^{r,2})$, where $v_i' = H(B_{\ell^r}^r)$. When $r = 0$, the only difference is that $\beta_i^{r,2} = t_2$ rather than being in a range. Similar things can be said for future steps and we will not emphasize them again.

Step 3. Arbitrarily fix an honest verifier $i \in HSV^{r,3}$. By Lemma 5.5, when player i is done waiting at time $\beta_i^{r,3} = \alpha_i^{r,3} + t_3$, he has received all messages sent by verifiers in $HSV^{r,2}$.

By the relationships among the parameters, with overwhelming probability $|HSV^{r,2}| > 2|MSV^{r,2}|$. Moreover, no honest verifier would sign contradicting messages, and the Adversary cannot forge a signature of an honest verifier after the latter has destroyed his corresponding ephemeral secret key. Thus more than 2/3 of all the valid $(r,2)$-messages i has received are from honest verifiers and of the form $m_j^{r,2} = (ESIG_j(H(B_{\ell^r}^r)), \sigma_j^{r,2})$, with no contradiction.

Accordingly, at time $\beta_i^{r,3}$ player i propagates $m_i^{r,3} = (ESIG_i(v'), \sigma_i^{r,3})$, where $v' = H(B_{\ell^r}^r)$.

Step 4. Arbitrarily fix an honest verifier $i \in HSV^{r,4}$. By Lemma 5.5, player i has received all messages sent by verifiers in $HSV^{r,3}$ when he is done waiting at time $\beta_i^{r,4} = \alpha_i^{r,4} + t_4$. Similar to Step 3, more than 2/3 of all the valid $(r,3)$-messages i has received are from honest verifiers and of the form $m_j^{r,3} = (ESIG_j(H(B_{\ell^r}^r)), \sigma_j^{r,3})$.

Accordingly, player i sets $v_i = H(B_{\ell^r}^r)$, $g_i = 2$ and $b_i = 0$. At time $\beta_i^{r,4} = \alpha_i^{r,4} + t_4$ he propagates $m_i^{r,4} = (ESIG_i(0), ESIG_i(H(B_{\ell^r}^r)), \sigma_i^{r,4})$.

Step 5. Arbitrarily fix an honest verifier $i \in HSV^{r,5}$. By Lemma 5.5, player i would have received all messages sent by the verifiers in $HSV^{r,4}$ if he has waited till time $\alpha_i^{r,5} + t_5$. Note that $|HSV^{r,4}| \geq t_H$.[19] Also note that all verifiers in $HSV^{r,4}$ have signed for $H(B_{\ell^r}^r)$.

As $|MSV^{r,4}| < t_H$, there does not exist any $v' \neq H(B_{\ell^r}^r)$ that could have been signed by t_H verifiers in $SV^{r,4}$ (who would necessarily be malicious), so player i does not stop before he has received t_H valid messages $m_j^{r,4} = (ESIG_j(0), ESIG_j(H(B_{\ell^r}^r)), \sigma_j^{r,4})$. Let T be the time when the latter event happens. Some of those messages may be from malicious players, but because $|MSV^{r,4}| < t_H$, at least one of them is from an honest verifier in $HSV^{r,4}$ and is sent after time $T^r + t_4$. Accordingly, $T \geq T^r + t_4 > T^r + \lambda + \Lambda \geq \beta_{\ell^r}^{r,1} + \Lambda$, and by time T player i has also received the message $m_{\ell^r}^{r,1}$. By the construction of the protocol, player i stops at time $\beta_i^{r,5} = T$ without propagating anything; sets $B^r = B_{\ell^r}^r$; and sets his own $CERT^r$ to be the set of $(r,4)$-messages for 0 and $H(B_{\ell^r}^r)$ that he has received.

Step $s > 5$. Similarly, for any step $s > 5$ and any verifier $i \in HSV^{r,s}$, player i would have received all messages sent by the verifiers in $HSV^{r,4}$ if he has waited till time $\alpha_i^{r,s} + t_s$. By the same analysis, player i stops without propagating anything, setting $B^r = B_{\ell^r}^r$ (and setting his own $CERT^r$ properly). Of course, the malicious verifiers may not stop and may propagate arbitrary messages, but because $|MSV^{r,s}| < t_H$, by induction no other v' could be signed by t_H verifiers in any step $4 \leq s' < s$, thus the honest verifiers only stop because they have received t_H valid $(r,4)$-messages for 0 and $H(B_{\ell^r}^r)$.

Reconstruction of the Round-r Block. The analysis of Step 5 applies to a generic honest user i almost without any change. Indeed, player i starts his own round r in the interval I^r and will only stop at a time T when he has received t_H valid $(r,4)$-messages for $H(B_{\ell^r}^r)$. Again because at least one of those messages are from honest verifiers and are sent after time $T^r + t_4$, player i has also received $m_{\ell^r}^{r,1}$ by time T. Thus he sets $B^r = B_{\ell^r}^r$ with the proper $CERT^r$.

It only remains to show that all honest users finish their round r within the time interval I^{r+1}. By the analysis of Step 5, every honest verifier $i \in HSV^{r,5}$ knows B^r on or before $\alpha_i^{r,5} + t_5 \leq T^r + \lambda + t_5 = T^r + 8\lambda + \Lambda$. Since T^{r+1} is the time when the first honest user i^r knows B^r, we have

$$T^{r+1} \leq T^r + 8\lambda + \Lambda$$

as desired. Moreover, when player i^r knows B^r, he has already helped propagating the messages in his $CERT^r$. Note that all those messages will be received by all honest users within time λ, even if

[19] Strictly speaking, this happens with very high probability but not necessarily overwhelming. However, this probability slightly effects the running time of the protocol, but does not affect its correctness. When $h = 80\%$, then $|HSV^{r,4}| \geq t_H$ with probability $1 - 10^{-8}$. If this event does not occur, then the protocol will continue for another 3 steps. As the probability that this does not occur in two steps is negligible, the protocol will finish at Step 8. In expectation, then, the number of steps needed is almost 5.

player i^r were the first player to propagate them. Moreover, following the analysis above we have $T^{r+1} \geq T^r + t_4 \geq \beta^{r,1}_{\ell^r} + \Lambda$, thus all honest users have received $m^{r,1}_{\ell^r}$ by time $T^{r+1} + \lambda$. Accordingly, all honest users know B^r in the time interval $I^{r+1} = [T^{r+1}, T^{r+1} + \lambda]$.

Finally, for $r = 0$ we actually have $T^1 \leq t_4 + \lambda = 6\lambda + \Lambda$. Combining everything together, Lemma 5.2 holds. ■

5.8 The Soundness Lemma

Lemma 5.3. [Soundness Lemma, restated] *Assuming Properties 1–3 hold for round $r-1$, when the leader ℓ^r is malicious, with overwhelming probability, all honest users agree on the same block B^r, $T^{r+1} \leq T^r + (6L^r + 10)\lambda + \Lambda$ and all honest users know B^r in the time interval I^{r+1}.*

Proof. We consider the two parts of the protocol, GC and $BBA^\star$, separately.

GC. By the inductive hypothesis and by Lemma 5.5, for any step $s \in \{2, 3, 4\}$ and any honest verifier $i \in HSV^{r,s}$, when player i acts at time $\beta^{r,s}_i = \alpha^{r,s}_i + t_s$, he has received all messages sent by all the honest verifiers in steps $s' < s$. We distinguish two possible cases for step 4.

Case 1. *No verifier $i \in HSV^{r,4}$ sets $g_i = 2$.*

In this case, by definition $b_i = 1$ for all verifiers $i \in HSV^{r,4}$. That is, they start with an agreement on 1 in the binary BA protocol. They may not have an agreement on their v_i's, but this does not matter as we will see in the binary BA.

Case 2. *There exists a verifier $\hat{i} \in HSV^{r,4}$ such that $g_{\hat{i}} = 2$.*

In this case, we show that

(1) $g_i \geq 1$ for all $i \in HSV^{r,4}$,

(2) there exists a value v' such that $v_i = v'$ for all $i \in HSV^{r,4}$, and

(3) there exists a valid message $m^{r,1}_\ell$ from some verifier $\ell \in SV^{r,1}$ such that $v' = H(B^r_\ell)$.

Indeed, since player $\hat{i}$ is honest and sets $g_{\hat{i}} = 2$, more than 2/3 of all the valid messages $m^{r,3}_j$ he has received are for the same value $v' \neq \perp$, and he has set $v_{\hat{i}} = v'$.

By Property (d) in Lemma 5.5, for any other honest $(r, 4)$-verifier i, it cannot be that more than 2/3 of all the valid messages $m^{r,3}_j$ that i' has received are for the same value $v'' \neq v'$. Accordingly, if i sets $g_i = 2$, it must be that i has seen $> 2/3$ majority for v' as well and set $v_i = v'$, as desired.

Now consider an arbitrary verifier $i \in HSV^{r,4}$ with $g_i < 2$. Similar to the analysis of Property (d) in Lemma 5.5, because player $\hat{i}$ has seen $> 2/3$ majority for v', more than $\frac{1}{2}|HSV^{r,3}|$ honest $(r, 3)$-verifiers have signed v'. Because i has received all messages by honest $(r, 3)$-verifiers by time $\beta^{r,4}_i = \alpha^{r,4}_i + t_4$, he has in particular received more than $\frac{1}{2}|HSV^{r,3}|$ messages from them for v'. Because $|HSV^{r,3}| > 2|MSV^{r,3}|$, i has seen $> 1/3$ majority for v'. Accordingly, player i sets $g_i = 1$, and Property (1) holds.

Does player i necessarily set $v_i = v'$? Assume there exists a different value $v'' \neq \perp$ such that player i has also seen $> 1/3$ majority for v''. Some of those messages may be from malicious verifiers, but at least one of them is from some honest verifier $j \in HSV^{r,3}$: indeed, because $|HSV^{r,3}| > 2|MSV^{r,3}|$ and i has received all messages from $HSV^{r,3}$, the set of malicious verifiers from whom i has received a valid $(r, 3)$-message counts for $< 1/3$ of all the valid messages he has received.

By definition, player j must have seen $> 2/3$ majority for v'' among all the valid $(r,2)$-messages he has received. However, we already have that some other honest $(r,3)$-verifiers have seen $> 2/3$ majority for v' (because they signed v'). By Property (d) of Lemma 5.5, this cannot happen and such a value v'' does not exist. Thus player i must have set $v_i = v'$ as desired, and Property (2) holds.

Finally, given that some honest $(r,3)$-verifiers have seen $> 2/3$ majority for v', some (actually, more than half of) honest $(r,2)$-verifiers have signed for v' and propagated their messages. By the construction of the protocol, those honest $(r,2)$-verifiers must have received a valid message $m_\ell^{r,1}$ from some player $\ell \in SV^{r,1}$ with $v' = H(B_\ell^r)$, thus Property (3) holds.

BBA⋆. We again distinguish two cases.

Case 1. *All verifiers $i \in HSV^{r,4}$ have $b_i = 1$.*

This happens following Case 1 of GC. As $|MSV^{r,4}| < t_H$, in this case no verifier in $SV^{r,5}$ could collect or generate t_H valid $(r,4)$-messages for bit 0. Thus, no honest verifier in $HSV^{r,5}$ would stop because he knows a non-empty block B^r.

Moreover, although there are at least t_H valid $(r,4)$-messages for bit 1, $s' = 5$ does not satisfy $s' - 2 \equiv 1 \mod 3$, thus no honest verifier in $HSV^{r,5}$ would stop because he knows $B^r = B_\epsilon^r$. Instead, every verifier $i \in HSV^{r,5}$ acts at time $\beta_i^{r,5} = \alpha_i^{r,5} + t_5$, by when he has received all messages sent by $HSV^{r,4}$ following Lemma 5.5. Thus player i has seen $> 2/3$ majority for 1 and sets $b_i = 1$.

In Step 6 which is a Coin-Fixed-To-1 step, although $s' = 5$ satisfies $s' - 2 \equiv 0 \mod 3$, there do not exist t_H valid $(r,4)$-messages for bit 0, thus no verifier in $HSV^{r,6}$ would stop because he knows a non-empty block B^r. However, with $s' = 6$, $s' - 2 \equiv 1 \mod 3$ and there do exist $|HSV^{r,5}| \geq t_H$ valid $(r,5)$-messages for bit 1 from $HSV^{r,5}$.

For every verifier $i \in HSV^{r,6}$, following Lemma 5.5, on or before time $\alpha_i^{r,6} + t_6$ player i has received all messages from $HSV^{r,5}$, thus i stops without propagating anything and sets $B^r = B_\epsilon^r$. His $CERT^r$ is the set of t_H valid $(r,5)$-messages $m_j^{r,5} = (ESIG_j(1), ESIG_j(v_j), \sigma_j^{r,5})$ received by him when he stops.

Next, let player i be either an honest verifier in a step $s > 6$ or a generic honest user (i.e., non-verifier). Similar to the proof of Lemma 5.2, player i sets $B^r = B_\epsilon^r$ and sets his own $CERT^r$ to be the set of t_H valid $(r,5)$-messages $m_j^{r,5} = (ESIG_j(1), ESIG_j(v_j), \sigma_j^{r,5})$ he has received.

Finally, similar to Lemma 5.2,

$$T^{r+1} \leq \min_{i \in HSV^{r,6}} \alpha_i^{r,6} + t_6 \leq T^r + \lambda + t_6 = T^r + 10\lambda + \Lambda,$$

and all honest users know B^r in the time interval I^{r+1}, because the first honest user i who knows B^r has helped propagating the $(r,5)$-messages in his $CERT^r$.

Case 2. *There exists a verifier $\hat{i} \in HSV^{r,4}$ with $b_{\hat{i}} = 0$.*

This happens following Case 2 of GC and is the more complex case. By the analysis of GC, in this case there exists a valid message $m_\ell^{r,1}$ such that $v_i = H(B_\ell^r)$ for all $i \in HSV^{r,4}$. Note that the verifiers in $HSV^{r,4}$ may not have an agreement on their b_i's.

For any step $s \in \{5, \ldots, m+3\}$ and verifier $i \in HSV^{r,s}$, by Lemma 5.5 player i would have received all messages sent by all honest verifiers in $HSV^{r,4} \cup \cdots \cup HSV^{r,s-1}$ if he has waited for time t_s.

We now consider the following event E: *there exists a step $s^* \geq 5$ such that, for the first time in the binary BA, some player $i^* \in SV^{r,s^*}$ (whether malicious or honest) should stop without propagating anything.* We use "should stop" to emphasize the fact that, if player i^* is malicious, then he may pretend that he should not stop according to the protocol and propagate messages of the Adversary's choice.

Moreover, by the construction of the protocol, either

($E.a$) i^* is able to collect or generate at least t_H valid messages $m_j^{r,s'-1} = (ESIG_j(0), ESIG_j(v), \sigma_j^{r,s'-1})$ for the same v and s', with $5 \leq s' \leq s^*$ and $s' - 2 \equiv 0 \mod 3$; or

($E.b$) i^* is able to collect or generate at least t_H valid messages $m_j^{r,s'-1} = (ESIG_j(1), ESIG_j(v_j), \sigma_j^{r,s'-1})$ for the same s', with $6 \leq s' \leq s^*$ and $s' - 2 \equiv 1 \mod 3$.

Because the honest $(r, s'-1)$-messages are received by all honest (r, s')-verifiers before they are done waiting in Step s', and because the Adversary receives everything no later than the honest users, without loss of generality we have $s' = s^*$ and player i^* is malicious. Note that we did not require the value v in $E.a$ to be the hash of a valid block: as it will become clear in the analysis, $v = H(B_\ell^r)$ in this sub-event.

Below we first analyze Case 2 following event E, and then show that the value of s^* is essentially distributed accordingly to L^r (thus event E happens before Step $m+3$ with overwhelming probability given the relationships for parameters). To begin with, for any step $5 \leq s < s^*$, every honest verifier $i \in HSV^{r,s}$ has waited time t_s and set v_i to be the majority vote of the valid $(r, s-1)$-messages he has received. Since player i has received all honest $(r, s-1)$-messages following Lemma 5.5, since all honest verifiers in $HSV^{r,4}$ have signed $H(B_\ell^r)$ following Case 2 of GC, and since $|HSV^{r,s-1}| > 2|MSV^{r,s-1}|$ for each s, by induction we have that player i has set

$$v_i = H(B_\ell^r).$$

The same holds for every honest verifier $i \in HSV^{r,s^*}$ who does not stop without propagating anything. Now we consider Step s^* and distinguish four subcases.

Case 2.1.a. *Event $E.a$ happens and there exists an honest verifier $i' \in HSV^{r,s^*}$ who should also stop without propagating anything.*

In this case, we have $s^* - 2 \equiv 0 \mod 3$ and Step s^* is a Coin-Fixed-To-0 step. By definition, player i' has received at least t_H valid (r, s^*-1)-messages of the form $(ESIG_j(0), ESIG_j(v), \sigma_j^{r,s^*-1})$. Since all verifiers in HSV^{r,s^*-1} have signed $H(B_\ell^r)$ and $|MSV^{r,s^*-1}| < t_H$, we have $v = H(B_\ell^r)$.

Since at least $t_H - |MSV^{r,s^*-1}| \geq 1$ of the (r, s^*-1)-messages received by i' for 0 and v are sent by verifiers in HSV^{r,s^*-1} after time $T^r + t_{s^*-1} \geq T^r + t_4 \geq T^r + \lambda + \Lambda \geq \beta_\ell^{r,1} + \Lambda$, player i' has received $m_\ell^{r,1}$ by the time he receives those (r, s^*-1)-messages. Thus player i' stops without propagating anything; sets $B^r = B_\ell^r$; and sets his own $CERT^r$ to be the set of valid (r, s^*-1)-messages for 0 and v that he has received.

Next, we show that, any other verifier $i \in HSV^{r,s^*}$ has either stopped with $B^r = B_\ell^r$, or has set $b_i = 0$ and propagated $(ESIG_i(0), ESIG_i(H(B_\ell^r)), \sigma_i^{r,s})$. Indeed, because Step s^* is the first time some verifier should stop without propagating anything, there does not exist a step $s' < s^*$ with $s' - 2 \equiv 1 \mod 3$ such that t_H $(r, s'-1)$-verifiers have signed 1. Accordingly, no verifier in HSV^{r,s^*} stops with $B^r = B_\epsilon^r$.

Moreover, as all honest verifiers in steps $\{4, 5, \ldots, s^* - 1\}$ have signed $H(B^r_\ell)$, there does not exist a step $s' \leq s^*$ with $s' - 2 \equiv 0 \mod 3$ such that t_H $(r, s'-1)$-verifiers have signed some $v'' \neq H(B^r_\ell)$ —indeed, $|MSV^{r,s'-1}| < t_H$. Accordingly, no verifier in HSV^{r,s^*} stops with $B^r \neq B^r_\epsilon$ and $B^r \neq B^r_\ell$. That is, if a player $i \in HSV^{r,s^*}$ has stopped without propagating anything, he must have set $B^r = B^r_\ell$.

If a player $i \in HSV^{r,s^*}$ has waited time t_{s^*} and propagated a message at time $\beta_i^{r,s^*} = \alpha_i^{r,s^*} + t_{s^*}$, he has received all messages from HSV^{r,s^*-1}, including at least $t_H - |MSV^{r,s^*-1}|$ of them for 0 and v. If i has seen $> 2/3$ majority for 1, then he has seen more than $2(t_H - |MSV^{r,s^*-1}|)$ valid $(r, s^* - 1)$-messages for 1, with more than $2t_H - 3|MSV^{r,s^*-1}|$ of them from honest $(r, s^* - 1)$-verifiers. However, this implies $|HSV^{r,s^*-1}| \geq t_H - |MSV^{r,s^*-1}| + 2t_H - 3|MSV^{r,s^*-1}| > 2n - 4|MSV^{r,s^*-1}|$, contradicting the fact that

$$|HSV^{r,s^*-1}| + 4|MSV^{r,s^*-1}| < 2n,$$

which comes from the relationships for the parameters. Accordingly, i does not see $> 2/3$ majority for 1, and he sets $b_i = 0$ because Step s^* is a Coin-Fixed-To-0 step. As we have seen, $v_i = H(B^r_\ell)$. Thus i propagates $(ESIG_i(0), ESIG_i(H(B^r_\ell)), \sigma_i^{r,s})$ as we wanted to show.

For Step $s^* + 1$, since player i' has helped propagating the messages in his $CERT^r$ on or before time $\alpha_{i'}^{r,s^*} + t_{s^*}$, all honest verifiers in HSV^{r,s^*+1} have received at least t_H valid $(r, s^* - 1)$-messages for bit 0 and value $H(B^r_\ell)$ on or before they are done waiting. Furthermore, verifiers in HSV^{r,s^*+1} will not stop before receiving those (r, s^*-1)-messages, because there do not exist any other t_H valid $(r, s' - 1)$-messages for bit 1 with $s' - 2 \equiv 1 \mod 3$ and $6 \leq s' \leq s^* + 1$, by the definition of Step s^*. In particular, Step $s^* + 1$ itself is a Coin-Fixed-To-1 step, but no honest verifier in HSV^{r,s^*} has propagated a message for 1, and $|MSV^{r,s^*}| < t_H$.

Thus all honest verifiers in HSV^{r,s^*+1} stop without propagating anything and set $B^r = B^r_\ell$: as before, they have received $m_\ell^{r,1}$ before they receive the desired (r, s^*-1)-messages.[20] The same can be said for all honest verifiers in future steps and all honest users in general. In particular, they all know $B^r = B^r_\ell$ within the time interval I^{r+1} and

$$T^{r+1} \leq \alpha_{i'}^{r,s^*} + t_{s^*} \leq T^r + \lambda + t_{s^*}.$$

Case 2.1.b. *Event E.b happens and there exists an honest verifier $i' \in HSV^{r,s^*}$ who should also stop without propagating anything.*

In this case we have $s^* - 2 \equiv 1 \mod 3$ and Step s^* is a Coin-Fixed-To-1 step. The analysis is similar to Case 2.1.a and many details have been omitted.

[20] If ℓ is malicious, he might send out $m_\ell^{r,1}$ late, hoping that some honest users/verifiers have not received $m_\ell^{r,1}$ yet when they receive the desired certificate for it. However, since verifier $\hat{i} \in HSV^{r,4}$ has set $b_{\hat{i}} = 0$ and $v_{\hat{i}} = H(B^r_\ell)$, as before we have that more than half of honest verifiers $i \in HSV^{r,3}$ have set $v_i = H(B^r_\ell)$. This further implies more than half of honest verifiers $i \in HSV^{r,2}$ have set $v_i = H(B^r_\ell)$, and those $(r, 2)$-verifiers have all received $m_\ell^{r,1}$. As the Adversary cannot distinguish a verifier from a non-verifier, he cannot target the propagation of $m_\ell^{r,1}$ to $(r, 2)$-verifiers without having the non-verifiers seeing it. In fact, with high probability, more than half (or a good constant fraction) of all honest users have seen $m_\ell^{r,1}$ after waiting for t_2 from the beginning of their own round r. From here on, the time λ' needed for $m_\ell^{r,1}$ to reach the remaining honest users is much smaller than Λ, and for simplicity we do not write it out in the analysis. If $4\lambda \geq \lambda'$ then the analysis goes through without any change: by the end of Step 4, all honest users would have received $m_\ell^{r,1}$. If the size of the block becomes enormous and $4\lambda < \lambda'$, then in Steps 3 and 4, the protocol could ask each verifier to wait for $\lambda'/2$ rather than 2λ, and the analysis continues to hold.

As before, player i' must have received at least t_H valid (r, s^*-1)-messages of the form $(ESIG_j(1), ESIG_j(v_j), \sigma_j^{r,s^*-1})$. Again by the definition of s^*, there does not exist a step $5 \le s' < s^*$ with $s' - 2 \equiv 0 \mod 3$, where at least t_H $(r, s'-1)$-verifiers have signed 0 and the same v. Thus player i' stops without propagating anything; sets $B^r = B^r_\epsilon$; and sets his own $CERT^r$ to be the set of valid (r, s^*-1)-messages for bit 1 that he has received.

Moreover, any other verifier $i \in HSV^{r,s^*}$ has either stopped with $B^r = B^r_\epsilon$, or has set $b_i = 1$ and propagated $(ESIG_i(1), ESIG_i(v_i), \sigma_i^{r,s^*})$. Since player i' has helped propagating the (r, s^*-1)-messages in his $CERT^r$ by time $\alpha_{i'}^{r,s^*} + t_{s^*}$, again all honest verifiers in HSV^{r,s^*+1} stop without propagating anything and set $B^r = B^r_\epsilon$. Similarly, all honest users know $B^r = B^r_\epsilon$ within the time interval I^{r+1} and

$$T^{r+1} \le \alpha_{i'}^{r,s^*} + t_{s^*} \le T^r + \lambda + t_{s^*}.$$

Case 2.2.a. *Event E.a happens and there does not exist an honest verifier $i' \in HSV^{r,s^*}$ who should also stop without propagating anything.*

In this case, note that player i^* could have a valid $CERT^r_{i^*}$ consisting of the t_H desired (r, s^*-1)-messages the Adversary is able to collect or generate. However, the malicious verifiers may not help propagating those messages, so we cannot conclude that the honest users will receive them in time λ. In fact, $|MSV^{r,s^*-1}|$ of those messages may be from malicious (r, s^*-1)-verifiers, who did not propagate their messages at all and only send them to the malicious verifiers in step s^*.

Similar to Case 2.1.a, here we have $s^* - 2 \equiv 0 \mod 3$, Step s^* is a Coin-Fixed-To-0 step, and the (r, s^*-1)-messages in $CERT^r_{i^*}$ are for bit 0 and $v = H(B^r_\ell)$. Indeed, all honest (r, s^*-1)-verifiers sign v, thus the Adversary cannot generate t_H valid (r, s^*-1)-messages for a different v'.

Moreover, all honest (r, s^*)-verifiers have waited time t_{s^*} and do not see $> 2/3$ majority for bit 1, again because $|HSV^{r,s^*-1}| + 4|MSV^{r,s^*-1}| < 2n$. Thus every honest verifier $i \in HSV^{r,s^*}$ sets $b_i = 0$, $v_i = H(B^r_\ell)$ by the majority vote, and propagates $m_i^{r,s^*} = (ESIG_i(0), ESIG_i(H(B^r_\ell)), \sigma_i^{r,s^*})$ at time $\alpha_i^{r,s^*} + t_{s^*}$.

Now consider the honest verifiers in Step s^*+1 (which is a Coin-Fixed-To-1 step). If the Adversary actually sends the messages in $CERT^r_{i^*}$ to some of them and causes them to stop, then similar to Case 2.1.a, all honest users know $B^r = B^r_\ell$ within the time interval I^{r+1} and

$$T^{r+1} \le T^r + \lambda + t_{s^*+1}.$$

Otherwise, all honest verifiers in Step s^*+1 have received all the (r, s^*)-messages for 0 and $H(B^r_\ell)$ from HSV^{r,s^*} after waiting time t_{s^*+1}, which leads to $> 2/3$ majority, because $|HSV^{r,s^*}| > 2|MSV^{r,s^*}|$. Thus all the verifiers in HSV^{r,s^*+1} propagate their messages for 0 and $H(B^r_\ell)$ accordingly. Note that the verifiers in HSV^{r,s^*+1} do not stop with $B^r = B^r_\ell$, because Step s^*+1 is not a Coin-Fixed-To-0 step.

Now consider the honest verifiers in Step s^*+2 (which is a Coin-Genuinely-Flipped step). If the Adversary sends the messages in $CERT^r_{i^*}$ to some of them and causes them to stop, then again all honest users know $B^r = B^r_\ell$ within the time interval I^{r+1} and

$$T^{r+1} \le T^r + \lambda + t_{s^*+2}.$$

Otherwise, all honest verifiers in Step s^*+2 have received all the (r, s^*+1)-messages for 0 and $H(B^r_\ell)$ from HSV^{r,s^*+1} after waiting time t_{s^*+2}, which leads to $> 2/3$ majority. Thus all of them propagate their messages for 0 and $H(B^r_\ell)$ accordingly: that is they do not "flip a coin" in this case. Again, note that they do not stop without propagating, because Step s^*+2 is not a Coin-Fixed-To-0 step.

Finally, for the honest verifiers in Step s^*+3 (which is another Coin-Fixed-To-0 step), all of them would have received at least t_H valid messages for 0 and $H(B^r_\ell)$ from HSV^{s^*+2}, if they really wait time t_{s^*+3}. Thus, whether or not the Adversary sends the messages in $CERT^r_{i^*}$ to any of them, all verifiers in HSV^{r,s^*+3} stop with $B^r = B^r_\ell$, without propagating anything. Depending on how the Adversary acts, some of them may have their own $CERT^r$ consisting of those (r, s^*-1)-messages in $CERT^r_{i^*}$, and the others have their own $CERT^r$ consisting of those (r, s^*+2)-messages. In any case, all honest users know $B^r = B^r_\ell$ within the time interval I^{r+1} and

$$T^{r+1} \leq T^r + \lambda + t_{s^*+3}.$$

Case 2.2.b. *Event E.b happens and there does not exist an honest verifier $i' \in HSV^{r,s^*}$ who should also stop without propagating anything.*
The analysis in this case is similar to those in Case 2.1.b and Case 2.2.a, thus many details have been omitted. In particular, $CERT^r_{i^*}$ consists of the t_H desired (r, s^*-1)-messages for bit 1 that the Adversary is able to collect or generate, $s^*-2 \equiv 1 \mod 3$, Step s^* is a Coin-Fixed-To-1 step, and no honest (r, s^*)-verifier could have seen $> 2/3$ majority for 0.

Thus, every verifier $i \in HSV^{r,s^*}$ sets $b_i = 1$ and propagates $m_i^{r,s^*} = (ESIG_i(1), ESIG_i(v_i), \sigma_i^{r,s^*})$ at time $\alpha_i^{r,s^*} + t_{s^*}$. Similar to Case 2.2.a, in at most 3 more steps (i.e., the protocol reaches Step s^*+3, which is another Coin-Fixed-To-1 step), all honest users know $B^r = B^r_\epsilon$ within the time interval I^{r+1}. Moreover, T^{r+1} may be $\leq T^r+\lambda+t_{s^*+1}$, or $\leq T^r+\lambda+t_{s^*+2}$, or $\leq T^r+\lambda+t_{s^*+3}$, depending on when is the first time an honest verifier is able to stop without propagating.

Combining the four sub-cases, we have that all honest users know B^r within the time interval I^{r+1}, with

$T^{r+1} \leq T^r + \lambda + t_{s^*}$ in Cases 2.1.a and 2.1.b, and

$T^{r+1} \leq T^r + \lambda + t_{s^*+3}$ in Cases 2.2.a and 2.2.b.

It remains to upper-bound s^* and thus T^{r+1} for Case 2, and we do so by considering how many times the Coin-Genuinely-Flipped steps are actually executed in the protocol: that is, some honest verifiers actually have flipped a coin.

In particular, arbitrarily fix a Coin-Genuinely-Flipped step s' (i.e., $7 \leq s' \leq m+2$ and $s'-2 \equiv 2 \mod 3$), and let $\ell' \triangleq \arg\min_{j \in SV^{r,s'-1}} H(\sigma_j^{r,s'-1})$. For now let us assume $s' < s^*$, because otherwise no honest verifier actually flips a coin in Step s', according to previous discussions.

By the definition of $SV^{r,s'-1}$, the hash value of the credential of ℓ' is also the smallest among all users in PK^{r-k}. Since the hash function is a random oracle, ideally player ℓ' is honest with probability at least h. As we will show later, even if the Adversary tries his best to predict the output of the random oracle and tilt the probability, player ℓ' is still honest with probability

at least $p_h = h^2(1 + h - h^2)$. Below we consider the case when that indeed happens: that is, $\ell' \in HSV^{r,s'-1}$.

Note that every honest verifier $i \in HSV^{r,s'}$ has received all messages from $HSV^{r,s'-1}$ by time $\alpha_i^{r,s'} + t_{s'}$. If player i needs to flip a coin (i.e., he has not seen $> 2/3$ majority for the same bit $b \in \{0,1\}$), then he sets $b_i = \mathtt{lsb}(H(\sigma_{\ell'}^{r,s'-1}))$. If there exists another honest verifier $i' \in HSV^{r,s'}$ who has seen $> 2/3$ majority for a bit $b \in \{0,1\}$, then by Property (d) of Lemma 5.5, no honest verifier in $HSV^{r,s'}$ would have seen $> 2/3$ majority for a bit $b' \neq b$. Since $\mathtt{lsb}(H(\sigma_{\ell'}^{r,s'-1})) = b$ with probability 1/2, all honest verifiers in $HSV^{r,s'}$ reach an agreement on b with probability 1/2. Of course, if such a verifier i' does not exist, then all honest verifiers in $HSV^{r,s'}$ agree on the bit $\mathtt{lsb}(H(\sigma_{\ell'}^{r,s'-1}))$ with probability 1.

Combining the probability for $\ell' \in HSV^{r,s'-1}$, we have that the honest verifiers in $HSV^{r,s'}$ reach an agreement on a bit $b \in \{0,1\}$ with probability at least $\frac{p_h}{2} = \frac{h^2(1+h-h^2)}{2}$. Moreover, by induction on the majority vote as before, all honest verifiers in $HSV^{r,s'}$ have their v_i's set to be $H(B_\ell^r)$. Thus, once an agreement on b is reached in Step s', T^{r+1} is

$$\text{either } \leq T^r + \lambda + t_{s'+1} \text{ or } \leq T^r + \lambda + t_{s'+2},$$

depending on whether $b = 0$ or $b = 1$, following the analysis of Cases 2.1.a and 2.1.b. In particular, no further Coin-Genuinely-Flipped step will be executed: that is, the verifiers in such steps still check that they are the verifiers and thus wait, but they will all stop without propagating anything. Accordingly, before Step s^*, the number of times the Coin-Genuinely-Flipped steps are executed is distributed according to the random variable L^r. Letting Step s' be the last Coin-Genuinely-Flipped step according to L^r, by the construction of the protocol we have

$$s' = 4 + 3L^r.$$

When should the Adversary make Step s^* happen if he wants to delay T^{r+1} as much as possible? We can even assume that the Adversary knows the realization of L^r in advance. If $s^* > s'$ then it is useless, because the honest verifiers have already reached an agreement in Step s'. To be sure, in this case s^* would be $s'+1$ or $s'+2$, again depending on whether $b = 0$ or $b = 1$. However, this is actually Cases 2.1.a and 2.1.b, and the resulting T^{r+1} is exactly the same as in that case. More precisely,

$$T^{r+1} \leq T^r + \lambda + t_{s^*} \leq T^r + \lambda + t_{s'+2}.$$

If $s^* < s' - 3$ —that is, s^* is before the second-last Coin-Genuinely-Flipped step— then by the analysis of Cases 2.2.a and 2.2.b,

$$T^{r+1} \leq T^r + \lambda + t_{s^*+3} < T^r + \lambda + t_{s'}.$$

That is, the Adversary is actually making the agreement on B^r happen faster.

If $s^* = s' - 2$ or $s' - 1$ —that is, the Coin-Fixed-To-0 step or the Coin-Fixed-To-1 step immediately before Step s'— then by the analysis of the four sub-cases, the honest verifiers in Step s' do not get to flip coins anymore, because they have either stopped without propagating, or have seen $> 2/3$ majority for the same bit b. Therefore we have

$$T^{r+1} \leq T^r + \lambda + t_{s^*+3} \leq T^r + \lambda + t_{s'+2}.$$

In sum, no matter what s^* is, we have

$$\begin{aligned} & T^{r+1} \leq T^r + \lambda + t_{s'+2} = T^r + \lambda + t_{3L^r+6} \\ = \; & T^r + \lambda + (2(3L^r + 6) - 3)\lambda + \Lambda \\ = \; & T^r + (6L^r + 10)\lambda + \Lambda, \end{aligned}$$

as we wanted to show. The worst case is when $s^* = s' - 1$ and Case 2.2.b happens.

Combining Cases 1 and 2 of the binary BA protocol, Lemma 5.3 holds. ■

5.9 Security of the Seed Q^r and Probability of An Honest Leader

It remains to prove Lemma 5.4. Recall that the verifiers in round r are taken from PK^{r-k} and are chosen according to the quantity Q^{r-1}. The reason for introducing the look-back parameter k is to make sure that, back at round $r-k$, when the Adversary is able to add new malicious users to PK^{r-k}, he cannot predict the quantity Q^{r-1} except with negligible probability. Note that the hash function is a random oracle and Q^{r-1} is one of its inputs when selecting verifiers for round r. Thus, no matter how malicious users are added to PK^{r-k}, from the Adversary's point of view each one of them is still selected to be a verifier in a step of round r with the required probability p (or p_1 for Step 1). More precisely, we have the following lemma.

Lemma 5.6. *With $k = O(\log_{1/2} F)$, for each round r, with overwhelming probability the Adversary did not query Q^{r-1} to the random oracle back at round $r-k$.*

Proof. We proceed by induction. Assume that for each round $\gamma < r$, the Adversary did not query $Q^{\gamma-1}$ to the random oracle back at round $\gamma - k$.[21] Consider the following mental game played by the Adversary at round $r-k$, trying to predict Q^{r-1}.

In Step 1 of each round $\gamma = r-k, \ldots, r-1$, given a specific $Q^{\gamma-1}$ not queried to the random oracle, by ordering the players $i \in PK^{\gamma-k}$ according to the hash values $H(SIG_i(\gamma, 1, Q^{\gamma-1}))$ increasingly, we obtain a random permutation over $PK^{\gamma-k}$. By definition, the leader ℓ^γ is the first user in the permutation and is honest with probability h. Moreover, when $PK^{\gamma-k}$ is large enough, for any integer $x \geq 1$, the probability that the first x users in the permutation are all malicious but the $(x+1)$st is honest is $(1-h)^x h$.

If ℓ^γ is honest, then $Q^\gamma = H(SIG_{\ell^\gamma}(Q^{\gamma-1}), \gamma)$. As the Adversary cannot forge the signature of ℓ^γ, Q^γ is distributed uniformly at random from the Adversary's point of view and, except with exponentially small probability,[22] was not queried to H at round $r-k$. Since each $Q^{\gamma+1}, Q^{\gamma+2}, \ldots, Q^{r-1}$ respectively is the output of H with $Q^\gamma, Q^{\gamma+1}, \ldots, Q^{r-2}$ as one of the inputs, they all look random to the Adversary and the Adversary could not have queried Q^{r-1} to H at round $r-k$.

Accordingly, the only case where the Adversary can predict Q^{r-1} with good probability at round $r-k$ is when all the leaders $\ell^{r-k}, \ldots, \ell^{r-1}$ are malicious. Again consider a round $\gamma \in \{r-k \ldots, r-1\}$ and the random permutation over $PK^{\gamma-k}$ induced by the corresponding hash values. If for some $x \geq 2$, the first $x-1$ users in the permutation are all malicious and the x-th is honest, then the Adversary has x possible choices for Q^γ: either of the form $H(SIG_i(Q^{\gamma-1}, \gamma))$, where i is one of

[21] As k is a small integer, without loss of generality one can assume that the first k rounds of the protocol are run under a safe environment and the inductive hypothesis holds for those rounds.

[22] That is, exponential in the length of the output of H. Note that this probability is way smaller than F.

the first $x-1$ malicious users, by making player i the actually leader of round γ; or $H(Q^{\gamma-1},\gamma)$, by forcing $B^\gamma = B_\epsilon^\gamma$. Otherwise, the leader of round γ will be the first honest user in the permutation and Q^{r-1} becomes unpredictable to the Adversary.

Which of the above x options of Q^γ should the Adversary pursue? To help the Adversary answer this question, in the mental game we actually make him more powerful than he actually is, as follows. First of all, in reality, the Adversary cannot compute the hash of a honest user's signature, thus cannot decide, for each Q^γ, the number $x(Q^\gamma)$ of malicious users at the beginning of the random permutation in round $\gamma+1$ induced by Q^γ. In the mental game, we give him the numbers $x(Q^\gamma)$ for free. Second of all, in reality, having the first x users in the permutation all being malicious does not necessarily mean they can all be made into the leader, because the hash values of their signatures must also be less than p_1. We have ignored this constraint in the mental game, giving the Adversary even more advantages.

It is easy to see that in the mental game, the optimal option for the Adversary, denoted by $\hat{Q}^\gamma$, is the one that produces the longest sequence of malicious users at the beginning of the random permutation in round $\gamma+1$. Indeed, given a specific Q^γ, the protocol does not depend on $Q^{\gamma-1}$ anymore and the Adversary can solely focus on the new permutation in round $\gamma+1$, which has the same distribution for the number of malicious users at the beginning. Accordingly, in each round γ, the above mentioned $\hat{Q}^\gamma$ gives him the largest number of options for $Q^{\gamma+1}$ and thus maximizes the probability that the consecutive leaders are all malicious.

Therefore, in the mental game the Adversary is following a Markov Chain from round $r-k$ to round $r-1$, with the state space being $\{0\}\cup\{x : x\geq 2\}$. State 0 represents the fact that the first user in the random permutation in the current round γ is honest, thus the Adversary fails the game for predicting Q^{r-1}; and each state $x\geq 2$ represents the fact that the first $x-1$ users in the permutation are malicious and the x-th is honest, thus the Adversary has x options for Q^γ. The transition probabilities $P(x,y)$ are as follows.

- $P(0,0)=1$ and $P(0,y)=0$ for any $y\geq 2$. That is, the Adversary fails the game once the first user in the permutation becomes honest.
- $P(x,0)=h^x$ for any $x\geq 2$. That is, with probability h^x, all the x random permutations have their first users being honest, thus the Adversary fails the game in the next round.
- For any $x\geq 2$ and $y\geq 2$, $P(x,y)$ is the probability that, among the x random permutations induced by the x options of Q^γ, the longest sequence of malicious users at the beginning of some of them is $y-1$, thus the Adversary has y options for $Q^{\gamma+1}$ in the next round. That is,

$$P(x,y)=\left(\sum_{i=0}^{y-1}(1-h)^i h\right)^x-\left(\sum_{i=0}^{y-2}(1-h)^i h\right)^x=(1-(1-h)^y)^x-(1-(1-h)^{y-1})^x.$$

Note that state 0 is the unique absorbing state in the transition matrix P, and every other state x has a positive probability of going to 0. We are interested in upper-bounding the number k of rounds needed for the Markov Chain to converge to 0 with overwhelming probability: that is, no matter which state the chain starts at, with overwhelming probability the Adversary loses the game and fails to predict Q^{r-1} at round $r-k$.

Consider the transition matrix $P^{(2)}\triangleq P\cdot P$ after two rounds. It is easy to see that $P^{(2)}(0,0)=1$ and $P^{(2)}(0,x)=0$ for any $x\geq 2$. For any $x\geq 2$ and $y\geq 2$, as $P(0,y)=0$, we have

$$P^{(2)}(x,y)=P(x,0)P(0,y)+\sum_{z\geq 2}P(x,z)P(z,y)=\sum_{z\geq 2}P(x,z)P(z,y).$$

Letting $\bar{h} \triangleq 1-h$, we have

$$P(x,y) = (1-\bar{h}^y)^x - (1-\bar{h}^{y-1})^x$$

and

$$P^{(2)}(x,y) = \sum_{z\geq 2}[(1-\bar{h}^z)^x - (1-\bar{h}^{z-1})^x][(1-\bar{h}^y)^z - (1-\bar{h}^{y-1})^z].$$

Below we compute the limit of $\frac{P^{(2)}(x,y)}{P(x,y)}$ as h goes to 1 —that is, $\bar{h}$ goes to 0. Note that the highest order of $\bar{h}$ in $P(x,y)$ is $\bar{h}^{y-1}$, with coefficient x. Accordingly,

$$\begin{aligned}
&\lim_{h\to 1}\frac{P^{(2)}(x,y)}{P(x,y)} = \lim_{\bar{h}\to 0}\frac{P^{(2)}(x,y)}{P(x,y)} = \lim_{\bar{h}\to 0}\frac{P^{(2)}(x,y)}{x\bar{h}^{y-1}+O(\bar{h}^y)}\\
= &\lim_{\bar{h}\to 0}\frac{\sum_{z\geq 2}[x\bar{h}^{z-1}+O(\bar{h}^z)][z\bar{h}^{y-1}+O(\bar{h}^y)]}{x\bar{h}^{y-1}+O(\bar{h}^y)} = \lim_{\bar{h}\to 0}\frac{2x\bar{h}^y+O(\bar{h}^{y+1})}{x\bar{h}^{y-1}+O(\bar{h}^y)}\\
= &\lim_{\bar{h}\to 0}\frac{2x\bar{h}^y}{x\bar{h}^{y-1}} = \lim_{\bar{h}\to 0} 2\bar{h} = 0.
\end{aligned}$$

When h is sufficiently close to 1,[23] we have

$$\frac{P^{(2)}(x,y)}{P(x,y)} \leq \frac{1}{2}$$

for any $x\geq 2$ and $y\geq 2$. By induction, for any $k>2$, $P^{(k)} \triangleq P^k$ is such that

- $P^{(k)}(0,0)=1$, $P^{(k)}(0,x)=0$ for any $x\geq 2$, and
- for any $x\geq 2$ and $y\geq 2$,

$$\begin{aligned}
&P^{(k)}(x,y) = P^{(k-1)}(x,0)P(0,y) + \sum_{z\geq 2}P^{(k-1)}(x,z)P(z,y) = \sum_{z\geq 2}P^{(k-1)}(x,z)P(z,y)\\
\leq &\sum_{z\geq 2}\frac{P(x,z)}{2^{k-2}}\cdot P(z,y) = \frac{P^{(2)}(x,y)}{2^{k-2}} \leq \frac{P(x,y)}{2^{k-1}}.
\end{aligned}$$

As $P(x,y)\leq 1$, after $1-\log_2 F$ rounds, the transition probability into any state $y\geq 2$ is negligible, starting with any state $x\geq 2$. Although there are many such states y, it is easy to see that

$$\lim_{y\to+\infty}\frac{P(x,y)}{P(x,y+1)} = \lim_{y\to+\infty}\frac{(1-\bar{h}^y)^x-(1-\bar{h}^{y-1})^x}{(1-\bar{h}^{y+1})^x-(1-\bar{h}^y)^x} = \lim_{y\to+\infty}\frac{\bar{h}^{y-1}-\bar{h}^y}{\bar{h}^y-\bar{h}^{y+1}} = \frac{1}{\bar{h}} = \frac{1}{1-h}.$$

Therefore each row x of the transition matrix P decreases as a geometric sequence with rate $\frac{1}{1-h}>2$ when y is large enough, and the same holds for $P^{(k)}$. Accordingly, when k is large enough but still on the order of $\log_{1/2} F$, $\sum_{y\geq 2}P^{(k)}(x,y) < F$ for any $x\geq 2$. That is, with overwhelming probability the Adversary loses the game and fails to predict Q^{r-1} at round $r-k$. For $h\in(2/3,1]$, a more complex analysis shows that there exists a constant C slightly larger than $1/2$, such that it suffices to take $k = O(\log_C F)$. Thus Lemma 5.6 holds. ■

Lemma 5.4. (restated) *Given Properties 1–3 for each round before r, $p_h = h^2(1+h-h^2)$ for L^r, and the leader ℓ^r is honest with probability at least p_h.*

[23] For example, $h = 80\%$ as suggested by the specific choices of parameters.

Proof. Following Lemma 5.6, the Adversary cannot predict Q^{r-1} back at round $r-k$ except with negligible probability. Note that this does not mean the probability of an honest leader is h for each round. Indeed, given Q^{r-1}, depending on how many malicious users are at the beginning of the random permutation of PK^{r-k}, the Adversary may have more than one options for Q^r and thus can increase the probability of a malicious leader in round $r+1$ —again we are giving him some unrealistic advantages as in Lemma 5.6, so as to simplify the analysis.

However, for each Q^{r-1} that was not queried to H by the Adversary back at round $r-k$, for any $x \geq 1$, with probability $(1-h)^{x-1}h$ the first honest user occurs at position x in the resulting random permutation of PK^{r-k}. When $x=1$, the probability of an honest leader in round $r+1$ is indeed h; while when $x=2$, the Adversary has two options for Q^r and the resulting probability is h^2. Only by considering these two cases, we have that the probability of an honest leader in round $r+1$ is at least $h \cdot h + (1-h)h \cdot h^2 = h^2(1+h-h^2)$ as desired.

Note that the above probability only considers the randomness in the protocol from round $r-k$ to round r. When all the randomness from round 0 to round r is taken into consideration, Q^{r-1} is even less predictable to the Adversary and the probability of an honest leader in round $r+1$ is at least $h^2(1+h-h^2)$. Replacing $r+1$ with r and shifts everything back by one round, the leader ℓ^r is honest with probability at least $h^2(1+h-h^2)$, as desired.

Similarly, in each Coin-Genuinely-Flipped step s, the "leader" of that step —that is the verifier in $SV^{r,s}$ whose credential has the smallest hash value, is honest with probability at least $h^2(1+h-h^2)$. Thus $p_h = h^2(1+h-h^2)$ for L^r and Lemma 5.4 holds. ■

6 *Algorand'$_2$*

In this section, we construct a version of *Algorand'* working under the following assumption.

Honest Majority of Users Assumption: *More than 2/3 of the users in each PK^r are honest.*

In Section 8, we show how to replace the above assumption with the desired Honest Majority of Money assumption.

6.1 Additional Notations and Parameters for *Algorand'$_2$*

Notations

- $\mu \in \mathbb{Z}^+$: a pragmatic upper-bound to the number of steps that, with overwhelming probability, will actually taken in one round. (As we shall see, parameter μ controls how many ephemeral keys a user prepares in advance for each round.)
- L^r: a random variable representing the number of Bernoulli trials needed to see a 1, when each trial is 1 with probability $\frac{p_h}{2}$. L^r will be used to upper-bound the time needed to generate block B^r.
- t_H: a lower-bound for the number of honest verifiers in a step $s>1$ of round r, such that with overwhelming probability (given n and p), there are $> t_H$ honest verifiers in $SV^{r,s}$.

Parameters

- *Relationships among various parameters.*
  - For each step $s>1$ of round r, n is chosen so that, with overwhelming probability,

$$|HSV^{r,s}| > t_H \quad \text{and} \quad |HSV^{r,s}| + 2|MSV^{r,s}| < 2t_H.$$

Note that the two inequalities above together imply $|HSV^{r,s}| > 2|MSV^{r,s}|$: that is, there is a 2/3 honest majority among selected verifiers.

The closer to 1 the value of h is, the smaller n needs to be. In particular, we use (variants of) Chernoff bounds to ensure the desired conditions hold with overwhelming probability.

- *Example choices of important parameters.*
  - $F = 10^{-18}$.
  - $n \approx 4000$, $t_H \approx 0.69n$, $k = 70$.

6.2 Implementing Ephemeral Keys in *Algorand*$'_2$

Recall that a verifier $i \in SV^{r,s}$ digitally signs his message $m_i^{r,s}$ of step s in round r, relative to an ephemeral public key $pk_i^{r,s}$, using an ephemeral secrete key $sk_i^{r,s}$ that he promptly destroys after using. When the number of possible steps that a round may take is capped by a given integer μ, we have already seen how to practically handle ephemeral keys. For example, as we have explained in *Algorand*$'_1$ (where $\mu = m + 3$), to handle all his possible ephemeral keys, from a round r' to a round $r' + 10^6$, i generates a pair (PMK, SMK), where PMK public master key of an identity based signature scheme, and SMK its corresponding secret master key. User i publicizes PMK and uses SMK to generate the secret key of each possible ephemeral public key (and destroys SMK after having done so). The set of i's ephemeral public keys for the relevant rounds is $S = \{i\} \times \{r', \ldots, r' + 10^6\} \times \{1, \ldots, \mu\}$. (As discussed, as the round $r' + 10^6$ approaches, i "refreshes" his pair (PMK, SMK).)

In practice, if μ is large enough, a round of *Algorand*$'_2$ will not take more than μ steps. In principle, however, there is the remote possibility that, for some round r the number of steps actually taken will exceed μ. When this happens, i would be unable to sign his message $m_i^{r,s}$ for any step $s > \mu$, because he has prepared in advance only μ secret keys for round r. Moreover, he could not prepare and publicize a new stash of ephemeral keys, as discussed before. In fact, to do so, he would need to insert a new public master key PMK' in a new block. But, should round r take more and more steps, no new blocks would be generated.

However, solutions exist. For instance, i may use the last ephemeral key of round r, $pk_i^{r,\mu}$, as follows. He generates another stash of key-pairs for round r —e.g., by (1) generating another master key pair $(\overline{PMK}, \overline{SMK})$; (2) using this pair to generate another, say, 10^6 ephemeral keys, $\overline{sk}_i^{r,\mu+1}, \ldots, \overline{sk}_i^{r,\mu+10^6}$, corresponding to steps $\mu+1, ..., \mu+10^6$ of round r; (3) using $sk_i^{r,\mu}$ to digitally sign $\overline{PMK}$ (and any (r,μ)-message if $i \in SV^{r,\mu}$), relative to $pk_i^{r,\mu}$; and (4) erasing $\overline{SMK}$ and $sk_i^{r,\mu}$. Should i become a verifier in a step $\mu + s$ with $s \in \{1, \ldots, 10^6\}$, then i digitally signs his $(r, \mu + s)$-message $m_i^{r,\mu+s}$ relative to his new key $\overline{pk}_i^{r,\mu+s} = (i, r, \mu + s)$. Of course, to verify this signature of i, others need to be certain that this public key corresponds to i's new public master key $\overline{PMK}$. Thus, in addition to this signature, i transmits his digital signature of $\overline{PMK}$ relative to $pk_i^{r,\mu}$.

Of course, this approach can be repeated, as many times as necessary, should round r continue for more and more steps! The last ephemeral secret key is used to authenticate a new master public key, and thus another stash of ephemeral keys for round r. And so on.

6.3 The Actual Protocol *Algorand'*$_2$

Recall again that, in each step s of a round r, a verifier $i \in SV^{r,s}$ uses his long-term public-secret key pair to produce his credential, $\sigma_i^{r,s} \triangleq SIG_i(r, s, Q^{r-1})$, as well as $SIG_i\left(Q^{r-1}\right)$ in case $s = 1$. Verifier i uses his ephemeral key pair, $(pk_i^{r,s}, sk_i^{r,s})$, to sign any other message m that may be required. For simplicity, we write $esig_i(m)$, rather than $sig_{pk_i^{r,s}}(m)$, to denote i's proper ephemeral signature of m in this step, and write $ESIG_i(m)$ instead of $SIG_{pk_i^{r,s}}(m) \triangleq (i, m, esig_i(m))$.

Step 1: Block Proposal

Instructions for every user $i \in PK^{r-k}$: User i starts his own Step 1 of round r as soon as he has $CERT^{r-1}$, which allows i to unambiguously compute $H(B^{r-1})$ and Q^{r-1}.

- User i uses Q^{r-1} to check whether $i \in SV^{r,1}$ or not. If $i \notin SV^{r,1}$, he does nothing for Step 1.
- If $i \in SV^{r,1}$, that is, if i is a potential leader, then he does the following.

  (a) If i has seen $B^0, \ldots, B^{r-1}$ himself (any $B^j = B_\epsilon^j$ can be easily derived from its hash value in $CERT^j$ and is thus assumed "seen"), then he collects the round-r payments that have been propagated to him so far and computes a maximal payset PAY_i^r from them.

  (b) If i hasn't seen all $B^0, \ldots, B^{r-1}$ yet, then he sets $PAY_i^r = \emptyset$.

  (c) Next, i computes his "candidate block" $B_i^r = (r, PAY_i^r, SIG_i(Q^{r-1}), H(B^{r-1}))$.

  (c) Finally, i computes the message $m_i^{r,1} = (B_i^r, esig_i(H(B_i^r)), \sigma_i^{r,1})$, destroys his ephemeral secret key $sk_i^{r,1}$, and then propagates two messages, $m_i^{r,1}$ and $(SIG_i(Q^{r-1}), \sigma_i^{r,1})$, separately but simultaneously.[a]

[a] When i is the leader, $SIG_i(Q^{r-1})$ allows others to compute $Q^r = H(SIG_i(Q^{r-1}), r)$.

Selective Propagation

To shorten the global execution of Step 1 and the whole round, it is important that the $(r, 1)$-messages are *selectively propagated*. That is, for every user j in the system,

- For the first $(r, 1)$-message that he ever receives and successfully verifies,[a] whether it contains a block or is just a credential and a signature of Q^{r-1}, player j propagates it as usual.
- For all the other $(r, 1)$-messages that player j receives and successfully verifies, he propagates it only if the hash value of the credential it contains is the *smallest* among the hash values of the credentials contained in all $(r, 1)$-messages he has received and successfully verified so far.
- However, if j receives two different messages of the form $m_i^{r,1}$ from the same player i,[b] he discards the second one no matter what the hash value of i's credential is.

Note that, under selective propagation it is useful that each potential leader i propagates his credential $\sigma_i^{r,1}$ separately from $m_i^{r,1}$:[c] those small messages travel faster than blocks, ensure timely propagation of the $m_i^{r,1}$'s where the contained credentials have small hash values, while make those with large hash values disappear quickly.

[a] That is, all the signatures are correct and, if it is of the form $m_i^{r,1}$, both the block and its hash are valid —although j does not check whether the included payset is maximal for i or not.

[b] Which means i is malicious.

[c] We thank Georgios Vlachos for suggesting this.

Step 2: The First Step of the Graded Consensus Protocol GC

Instructions for every user $i \in PK^{r-k}$: User i starts his own Step 2 of round r as soon as he has $CERT^{r-1}$.

- User i waits a maximum amount of time $t_2 \triangleq \lambda + \Lambda$. While waiting, i acts as follows.
  1. After waiting for time 2λ, he finds the user ℓ such that $H(\sigma_\ell^{r,1}) \leq H(\sigma_j^{r,1})$ for all credentials $\sigma_j^{r,1}$ that are part of the successfully verified $(r,1)$-messages he has received so far.[a]
  2. If he has received a block B^{r-1}, which matches the hash value $H(B^{r-1})$ contained in $CERT^{r-1}$,[b] and if he has received from ℓ a valid message $m_\ell^{r,1} = (B_\ell^r, esig_\ell(H(B_\ell^r)), \sigma_\ell^{r,1})$,[c] then i stops waiting and sets $v_i' \triangleq (H(B_\ell^r), \ell)$.
  3. Otherwise, when time t_2 runs out, i sets $v_i' \triangleq \perp$.
  4. When the value of v_i' has been set, i computes Q^{r-1} from $CERT^{r-1}$ and checks whether $i \in SV^{r,2}$ or not.
  5. If $i \in SV^{r,2}$, i computes the message $m_i^{r,2} \triangleq (ESIG_i(v_i'), \sigma_i^{r,2})$,[d] destroys his ephemeral secret key $sk_i^{r,2}$, and then propagates $m_i^{r,2}$. Otherwise, i stops without propagating anything.

[a] Essentially, user i privately decides that the leader of round r is user ℓ.

[b] Of course, if $CERT^{r-1}$ indicates that $B^{r-1} = B_\epsilon^{r-1}$, then i has already "received" B^{r-1} the moment he has $CERT^{r-1}$.

[c] Again, player ℓ's signatures and the hashes are all successfully verified, and PAY_ℓ^r in B_ℓ^r is a valid payset for round r —although i does not check whether PAY_ℓ^r is maximal for ℓ or not. If B_ℓ^r contains an empty payset, then there is actually no need for i to see B^{r-1} before verifying whether B_ℓ^r is valid or not.

[d] The message $m_i^{r,2}$ signals that player i considers the first component of v_i' to be the hash of the next block, or considers the next block to be empty.

Step 3: The Second Step of GC

Instructions for every user $i \in PK^{r-k}$: User i starts his own Step 3 of round r as soon as he has $CERT^{r-1}$.

- User i waits a maximum amount of time $t_3 \triangleq t_2 + 2\lambda = 3\lambda + \Lambda$. While waiting, i acts as follows.
  1. If there exists a value v such that he has received at least t_H valid messages $m_j^{r,2}$ of the form $(ESIG_j(v), \sigma_j^{r,2})$, without any contradiction,[a] then he stops waiting and sets $v' = v$.
  2. Otherwise, when time t_3 runs out, he sets $v' = \perp$.
  3. When the value of v' has been set, i computes Q^{r-1} from $CERT^{r-1}$ and checks whether $i \in SV^{r,3}$ or not.
  4. If $i \in SV^{r,3}$, then i computes the message $m_i^{r,3} \triangleq (ESIG_i(v'), \sigma_i^{r,3})$, destroys his ephemeral secret key $sk_i^{r,3}$, and then propagates $m_i^{r,3}$. Otherwise, i stops without propagating anything.

[a] That is, he has not received two valid messages containing $ESIG_j(v)$ and a different $ESIG_j(\hat{v})$ respectively, from a player j. Here and from here on, except in the Ending Conditions defined later, whenever an honest player wants messages of a given form, messages contradicting each other are never counted or considered valid.

Step 4: Output of GC and The First Step of $BBA^\star$

Instructions for every user $i \in PK^{r-k}$: User i starts his own Step 4 of round r as soon as he finishes his own Step 3.

- User i waits a maximum amount of time 2λ.[a] While waiting, i acts as follows.
  1. He computes v_i and g_i, the output of GC, as follows.
     (a) If there exists a value $v' \neq \perp$ such that he has received at least t_H valid messages $m_j^{r,3} = (ESIG_j(v'), \sigma_j^{r,3})$, then he stops waiting and sets $v_i \triangleq v'$ and $g_i \triangleq 2$.
     (b) If he has received at least t_H valid messages $m_j^{r,3} = (ESIG_j(\perp), \sigma_j^{r,3})$, then he stops waiting and sets $v_i \triangleq \perp$ and $g_i \triangleq 0$.[b]
     (c) Otherwise, when time 2λ runs out, if there exists a value $v' \neq \perp$ such that he has received at least $\lceil \frac{t_H}{2} \rceil$ valid messages $m_j^{r,j} = (ESIG_j(v'), \sigma_j^{r,3})$, then he sets $v_i \triangleq v'$ and $g_i \triangleq 1$.[c]
     (d) Else, when time 2λ runs out, he sets $v_i \triangleq \perp$ and $g_i \triangleq 0$.
  2. When the values v_i and g_i have been set, i computes b_i, the input of $BBA^\star$, as follows: $b_i \triangleq 0$ if $g_i = 2$, and $b_i \triangleq 1$ otherwise.
  3. i computes Q^{r-1} from $CERT^{r-1}$ and checks whether $i \in SV^{r,4}$ or not.
  4. If $i \in SV^{r,4}$, he computes the message $m_i^{r,4} \triangleq (ESIG_i(b_i), ESIG_i(v_i), \sigma_i^{r,4})$, destroys his ephemeral secret key $sk_i^{r,4}$, and propagates $m_i^{r,4}$. Otherwise, i stops without propagating anything.

[a] Thus, the maximum *total* amount of time since i starts his Step 1 of round r could be $t_4 \triangleq t_3 + 2\lambda = 5\lambda + \Lambda$.

[b] Whether Step (b) is in the protocol or not does not affect its correctness. However, the presence of Step (b) allows Step 4 to end in less than 2λ time if sufficiently many Step-3 verifiers have "signed $\perp$."

[c] It can be proved that the v' in this case, if exists, must be unique.

Step s, $5 \leq s \leq m+2$, $s-2 \equiv 0 \mod 3$: A Coin-Fixed-To-0 Step of $BBA^\star$

Instructions for every user $i \in PK^{r-k}$: User i starts his own Step s of round r as soon as he finishes his own Step $s-1$.

- User i waits a maximum amount of time 2λ.[a] While waiting, i acts as follows.
  - *Ending Condition 0:* If at any point there exists a string $v \neq \perp$ and a step s' such that
    (a) $5 \leq s' \leq s$, $s'-2 \equiv 0 \mod 3$ —that is, Step s' is a Coin-Fixed-To-0 step,
    (b) i has received at least t_H valid messages $m_j^{r,s'-1} = (ESIG_j(0), ESIG_j(v), \sigma_j^{r,s'-1})$,[b] and
    (c) i has received a valid message $(SIG_j(Q^{r-1}), \sigma_j^{r,1})$ with j being the second component of v,

    then, i stops waiting and ends his own execution of Step s (and in fact of round r) right away without propagating anything as a (r,s)-verifier; sets $H(B^r)$ to be the first component of v; and sets his own $CERT^r$ to be the set of messages $m_j^{r,s'-1}$ of step (b) together with $(SIG_j(Q^{r-1}), \sigma_j^{r,1})$.[c]
  - *Ending Condition 1:* If at any point there exists a step s' such that
    (a') $6 \leq s' \leq s$, $s'-2 \equiv 1 \mod 3$ —that is, Step s' is a Coin-Fixed-To-1 step, and
    (b') i has received at least t_H valid messages $m_j^{r,s'-1} = (ESIG_j(1), ESIG_j(v_j), \sigma_j^{r,s'-1})$,[d]

    then, i stops waiting and ends his own execution of Step s (and in fact of round r) right away without propagating anything as a (r,s)-verifier; sets $B^r = B^r_\epsilon$; and sets his own $CERT^r$ to be the set of messages $m_j^{r,s'-1}$ of sub-step (b').
  - If at any point he has received at least t_H valid $m_j^{r,s-1}$'s of the form $(ESIG_j(1), ESIG_j(v_j), \sigma_j^{r,s-1})$, then he stops waiting and sets $b_i \triangleq 1$.
  - If at any point he has received at least t_H valid $m_j^{r,s-1}$'s of the form $(ESIG_j(0), ESIG_j(v_j), \sigma_j^{r,s-1})$, but they do not agree on the same v, then he stops waiting and sets $b_i \triangleq 0$.
  - Otherwise, when time 2λ runs out, i sets $b_i \triangleq 0$.
  - When the value b_i has been set, i computes Q^{r-1} from $CERT^{r-1}$ and checks whether $i \in SV^{r,s}$.
  - If $i \in SV^{r,s}$, i computes the message $m_i^{r,s} \triangleq (ESIG_i(b_i), ESIG_i(v_i), \sigma_i^{r,s})$ with v_i being the value he has computed in Step 4, destroys his ephemeral secret key $sk_i^{r,s}$, and then propagates $m_i^{r,s}$. Otherwise, i stops without propagating anything.

[a] Thus, the maximum *total* amount of time since i starts his Step 1 of round r could be $t_s \triangleq t_{s-1} + 2\lambda = (2s-3)\lambda + \Lambda$.

[b] Such a message from player j is counted even if player i has also received a message from j signing for 1. Similar things for Ending Condition 1. As shown in the analysis, this is to ensure that all honest users know $CERT^r$ within time λ from each other.

[c] User i now knows $H(B^r)$ and his own round r finishes. He just needs to wait until the actually block B^r is propagated to him, which may take some additional time. He still helps propagating messages as a generic user, but does not initiate any propagation as a (r,s)-verifier. In particular, he has helped propagating all messages in his $CERT^r$, which is enough for our protocol. Note that he should also set $b_i \triangleq 0$ for the binary BA protocol, but b_i is not needed in this case anyway. Similar things for all future instructions.

[d] In this case, it does not matter what the v_j's are.

Step s, $6 \leq s \leq m+2$, $s-2 \equiv 1 \mod 3$: A Coin-Fixed-To-1 Step of $BBA^\star$

Instructions for every user $i \in PK^{r-k}$: User i starts his own Step s of round r as soon as he finishes his own Step $s-1$.

- User i waits a maximum amount of time 2λ. While waiting, i acts as follows.
  - *Ending Condition 0:* The same instructions as in a Coin-Fixed-To-0 step.
  - *Ending Condition 1:* The same instructions as in a Coin-Fixed-To-0 step.
  - If at any point he has received at least t_H valid $m_j^{r,s-1}$'s of the form $(ESIG_j(0), ESIG_j(v_j), \sigma_j^{r,s-1})$, then he stops waiting and sets $b_i \triangleq 0$.[a]
  - Otherwise, when time 2λ runs out, i sets $b_i \triangleq 1$.
  - When the value b_i has been set, i computes Q^{r-1} from $CERT^{r-1}$ and checks whether $i \in SV^{r,s}$.
  - If $i \in SV^{r,s}$, i computes the message $m_i^{r,s} \triangleq (ESIG_i(b_i), ESIG_i(v_i), \sigma_i^{r,s})$ with v_i being the value he has computed in Step 4, destroys his ephemeral secret key $sk_i^{r,s}$, and then propagates $m_i^{r,s}$. Otherwise, i stops without propagating anything.

[a] Note that receiving t_H valid $(r, s-1)$-messages signing for 1 would mean Ending Condition 1.

Step s, $7 \leq s \leq m+2$, $s-2 \equiv 2 \mod 3$: A Coin-Genuinely-Flipped Step of $BBA^\star$

Instructions for every user $i \in PK^{r-k}$: User i starts his own Step s of round r as soon as he finishes his own step $s-1$.

- User i waits a maximum amount of time 2λ. While waiting, i acts as follows.
  - *Ending Condition 0:* The same instructions as in a Coin-Fixed-To-0 step.
  - *Ending Condition 1:* The same instructions as in a Coin-Fixed-To-0 step.
  - If at any point he has received at least t_H valid $m_j^{r,s-1}$'s of the form $(ESIG_j(0), ESIG_j(v_j), \sigma_j^{r,s-1})$, then he stops waiting and sets $b_i \triangleq 0$.
  - If at any point he has received at least t_H valid $m_j^{r,s-1}$'s of the form $(ESIG_j(1), ESIG_j(v_j), \sigma_j^{r,s-1})$, then he stops waiting and sets $b_i \triangleq 1$.
  - Otherwise, when time 2λ runs out, letting $SV_i^{r,s-1}$ be the set of $(r, s-1)$-verifiers from whom he has received a valid message $m_j^{r,s-1}$, i sets $b_i \triangleq \mathtt{lsb}(\min_{j \in SV_i^{r,s-1}} H(\sigma_j^{r,s-1}))$.
  - When the value b_i has been set, i computes Q^{r-1} from $CERT^{r-1}$ and checks whether $i \in SV^{r,s}$.
  - If $i \in SV^{r,s}$, i computes the message $m_i^{r,s} \triangleq (ESIG_i(b_i), ESIG_i(v_i), \sigma_i^{r,s})$ with v_i being the value he has computed in Step 4, destroys his ephemeral secret key $sk_i^{r,s}$, and then propagates $m_i^{r,s}$. Otherwise, i stops without propagating anything.

Remark. In principle, as considered in subsection 6.2, the protocol may take arbitrarily many steps in some round. Should this happens, as discussed, a user $i \in SV^{r,s}$ with $s > \mu$ has exhausted

his stash of pre-generated ephemeral keys and has to authenticate his (r,s)-message $m_i^{r,s}$ by a "cascade" of ephemeral keys. Thus i's message becomes a bit longer and transmitting these longer messages will take a bit more time. Accordingly, after so many steps of a given round, the value of the parameter λ will automatically increase slightly. (But it reverts to the original λ once a new block is produced and a new round starts.)

Reconstruction of the Round-r Block by Non-Verifiers

Instructions for every user i in the system: User i starts his own round r as soon as he has $CERT^{r-1}$.

- i follows the instructions of each step of the protocol, participates the propagation of all messages, but does not initiate any propagation in a step if he is not a verifier in it.
- i ends his own round r by entering either Ending Condition 0 or Ending Condition 1 in some step, with the corresponding $CERT^r$.
- From there on, he starts his round $r+1$ while waiting to receive the actual block B^r (unless he has already received it), whose hash $H(B^r)$ has been pinned down by $CERT^r$. Again, if $CERT^r$ indicates that $B^r = B^r_\epsilon$, the i knows B^r the moment he has $CERT^r$.

6.4 Analysis of *Algorand'*$_2$

The analysis of *Algorand'*$_2$ is easily derived from that of *Algorand'*$_1$. Essentially, in *Algorand'*$_2$, with overwhelming probability, (a) all honest users agree on the same block B^r; the leader of a new block is honest with probability at least $p_h = h^2(1+h-h^2)$.

7 Handling Offline Honest users

As we said, a honest user follows all his prescribed instructions, which include that of being online and running the protocol. This is not a major burden in Algorand, since the computation and bandwidth required from a honest user are quite modest. Yet, let us point out that Algorand can be easily modified so as to work in two models, in which honest users are allowed to be offline in great numbers.

Before discussing these two models, let us point out that, if the percentage of honest players were 95%, Algorand could still be run setting all parameters assuming instead that $h = 80\%$. Accordingly, Algorand would continue to work properly even if at most half of the honest players chose to go offline (indeed, a major case of"absenteeism"). In fact, at any point in time, at least 80% of the players online would be honest.

From Continual Participation to Lazy Honesty As we saw, *Algorand'*$_1$ and *Algorand'*$_2$ choose the look-back parameter k. Let us now show that choosing k properly large enables one to remove the Continual Participation requirement. This requirement ensures a crucial property: namely, that the underlying BA protocol $BBA^\star$ has a proper honest majority. Let us now explain how lazy honesty provides an alternative and attractive way to satisfy this property.

Recall that a user i is lazy-but-honest if (1) he follows all his prescribed instructions, when he is asked to participate to the protocol, and (2) he is asked to participate to the protocol only very rarely —e.g., once a week— with suitable advance notice, and potentially receiving significant rewards when he participates.

To allow Algorand to work with such players, it just suffices to "choose the verifiers of the current round among the users already in the system in a much earlier round." Indeed, recall that the verifiers for a round r are chosen from users in round $r-k$, and the selections are made based on the quantity Q^{r-1}. Note that a week consists of roughly 10,000 minutes, and assume that a round takes roughly (e.g., on average) 5 minutes, so a week has roughly 2,000 rounds. Assume that, at some point of time, a user i wishes to plan his time and know whether he is going to be a verifier in the coming week. The protocol now chooses the verifiers for a round r from users in round $r-k-2,000$, and the selections are based on $Q^{r-2,001}$. At round r, player i already knows the values $Q^{r-2,000},\ldots,Q^{r-1}$, since they are actually part of the blockchain. Then, for each M between 1 and 2,000, i is a verifier in a step s of round $r+M$ if and only if

$$.H\left(SIG_i\left(r+M,s,Q^{r+M-2,001}\right)\right)\leq p\ .$$

Thus, to check whether he is going to be called to act as a verifier in the next 2,000 rounds, i must compute $\sigma_i^{M,s}=SIG_i\left(r+M,s,Q^{r+M-2,001}\right)$ for $M=1$ to $2,000$ and for each step s, and check whether $.H(\sigma_i^{M,s})\leq p$ for some of them. If computing a digital signature takes a millisecond, then this entire operation will take him about 1 minute of computation. If he is not selected as a verifier in any of these rounds, then he can go off-line with an "honest conscience". Had he continuously participated, he would have essentially taken 0 steps in the next 2,000 rounds anyway! If, instead, he is selected to be a verifier in one of these rounds, then he readies himself (e.g., by obtaining all the information necessary) to act as an honest verifier at the proper round.

By so acting, a lazy-but-honest potential verifier i only misses participating to the propagation of messages. But message propagation is typically robust. Moreover, the payers and the payees of recently propagated payments are expected to be online to watch what happens to their payments, and thus they will participate to message propagation, if they are honest.

8 Protocol *Algorand′* with Honest Majority of Money

We now, finally, show how to replace the Honest Majority of Users assumption with the much more meaningful Honest Majority of Money assumption. The basic idea is (in a proof-of-stake flavor) "to select a user $i\in PK^{r-k}$ to belong to $SV^{r,s}$ with a weight (i.e., decision power) proportional to the amount of money owned by i."[24]

By our HMM assumption, we can choose whether that amount should be owned at round $r-k$ or at (the start of) round r. Assuming that we do not mind continual participation, we opt for the latter choice. (To remove continual participation, we would have opted for the former choice. *Better said, for the amount of money owned at round* $r-k-2,000$.)

There are many ways to implement this idea. The simplest way would be to have each key hold at most 1 unit of money and then select at random n users i from PK^{r-k} such that $a_i^{(r)}=1$.

[24] We should say $PK^{r-k-2,000}$ so as to replace continual participation. For simplicity, since one may wish to require continual participation anyway, we use PK^{r-k} as before, so as to carry one less parameter.

The Next Simplest Implementation

The next simplest implementation may be to demand that each public key owns a maximum amount of money M, for some fixed M. The value M is small enough compared with the total amount of money in the system, such that the probability a key belongs to the verifier set of more than one step in —say— k rounds is negligible. Then, a key $i \in PK^{r-k}$, owning an amount of money $a_i^{(r)}$ in round r, is chosen to belong to $SV^{r,s}$ if

$$.H\left(SIG_i\left(r, s, Q^{r-1}\right)\right) \leq p \cdot \frac{a_i^{(r)}}{M} \ .$$

And all proceeds as before.

A More Complex Implementation

The last implementation "forced a rich participant in the system to own many keys".

An alternative implementation, described below, generalizes the notion of status and consider each user i to consist of $K+1$ *copies* (i, v), each of which is independently selected to be a verifier, and will own his own ephemeral key $(pk_{i,v}^{r,s}, sk_{i,v}^{r,s})$ in a step s of a round r. The value K depends on the amount of money $a_i^{(r)}$ owned by i in round r.

Let us now see how such a system works in greater detail.

Number of Copies Let n be the targeted expected cardinality of each verifier set, and let $a_i^{(r)}$ be the amount of money owned by a user i at round r. Let A^r be the total amount of money owned by the users in PK^{r-k} at round r, that is,

$$A^r = \sum_{i \in PK^{r-k}} a_i^{(r)}.$$

If i is an user in PK^{r-k}, then i's copies are $(i,1), \ldots, (i, K+1)$, where

$$K = \left\lfloor \frac{n \cdot a_i^{(r)}}{A^r} \right\rfloor \ .$$

Example. Let $n = 1,000$, $A^r = 10^9$, and $a_i^{(r)} = 3.7$ millions. Then,

$$K = \left\lfloor \frac{10^3 \cdot (3.7 \cdot 10^6)}{10^9} \right\rfloor = \lfloor 3.7 \rfloor = 3 \ .$$

Verifiers and Credentials Let i be a user in PK^{r-k} with $K+1$ copies.

For each $v = 1, \ldots, K$, copy (i, v) belongs to $SV^{r,s}$ automatically. That is, i's credential is $\sigma_{i,v}^{r,s} \triangleq SIG_i((i,v), r, s, Q^{r-1})$, but the corresponding condition becomes $.H(\sigma_{i,v}^{r,s}) \leq 1$, which is always true.

For copy $(i, K+1)$, for each Step s of round r, i checks whether

$$.H\big(SIG_i\big(\,(i, K+1), r, s, Q^{r-1}\,\big)\big) \leq a_i^{(r)} \frac{n}{A^r} - K \ .$$

If so, copy $(i, K+1)$ belongs to $SV^{r,s}$. To prove it, i propagates the credential

$$\sigma^{r,1}_{i,K+1} = SIG_i\big((i, K+1), r, s, Q^{r-1}\big).$$

EXAMPLE. As in the previous example, let $n = 1K$, $a_i^{(r)} = 3.7M$, $A^r = 1B$, and i has 4 copies: $(i,1), \ldots, (i,4)$. Then, the first 3 copies belong to $SV^{r,s}$ automatically. For the 4th one, conceptually, *Algorand′* independently rolls a biased coin, whose probability of Heads is 0.7. Copy $(i,4)$ is selected if and only if the coin toss is Heads.

(Of course, this biased coin flip is implemented by hashing, signing, and comparing —as we have done all along in this paper— so as to enable i to prove his result.)

Business as Usual Having explained how verifiers are selected and how their credentials are computed at each step of a round r, the execution of a round is similar to that already explained.

9 Handling Forks

Having reduced the probability of forks to 10^{-12} or 10^{-18}, it is practically unnecessary to handle them in the remote chance that they occur. Algorand, however, can also employ various fork resolution procedures, with or without proof of work.

One possible way of instructing the users to resolve forks is as follows:

- Follow the longest chain if a user sees multiple chains.
- If there are more than one longest chains, follow the one with a non-empty block at the end. If all of them have empty blocks at the end, consider their second-last blocks.
- If there are more than one longest chains with non-empty blocks at the end, say the chains are of length r, follow the one whose leader of block r has the smallest credential. If there are ties, follow the one whose block r itself has the smallest hash value. If there are still ties, follow the one whose block r is ordered the first lexicographically.

10 Handling Network Partitions

As said, we assume the propagation times of messages among all users in the network are upper-bounded by λ and Λ. This is not a strong assumption, as today's Internet is fast and robust, and the actual values of these parameters are quite reasonable. Here, let us point out that *Algorand′*$_2$ continues to work even if the Internet occasionally got partitioned into two parts. The case when the Internet is partitioned into more than two parts is similar.

10.1 Physical Partitions

First of all, the partition may be caused by physical reasons. For example, a huge earthquake may end up completely breaking down the connection between Europe and America. In this case, the malicious users are also partitioned and there is no communication between the two parts. Thus

there will be two Adversaries, one for part 1 and the other for part 2. Each Adversary still tries to break the protocol in its own part.

Assume the partition happens in the middle of round r. Then each user is still selected as a verifier based on PK^{r-k}, with the same probability as before. Let $HSV_i^{r,s}$ and $MSV_i^{r,s}$ respectively be the set of honest and malicious verifiers in a step s in part $i \in \{1,2\}$. We have

$$|HSV_1^{r,s}| + |MSV_1^{r,s}| + |HSV_2^{r,s}| + |MSV_2^{r,s}| = |HSV^{r,s}| + |MSV^{r,s}|.$$

Note that $|HSV^{r,s}| + |MSV^{r,s}| < |HSV^{r,s}| + 2|MSV^{r,s}| < 2t_H$ with overwhelming probability.

If some part i has $|HSV_i^{r,s}| + |MSV_i^{r,s}| \geq t_H$ with non-negligible probability, e.g., 1%, then the probability that $|HSV_{3-i}^{r,s}| + |MSV_{3-i}^{r,s}| \geq t_H$ is very low, e.g., 10^{-16} when $F = 10^{-18}$. In this case, we may as well treat the smaller part as going offline, because there will not be enough verifiers in this part to generate t_H signatures to certify a block.

Let us consider the larger part, say part 1 without loss of generality. Although $|HSV^{r,s}| < t_H$ with negligible probability in each step s, when the network is partitioned, $|HSV_1^{r,s}|$ may be less than t_H with some non-negligible probability. In this case the Adversary may, with some other non-negligible probability, force the binary BA protocol into a fork in round r, with a non-empty block B^r and the empty block B_ϵ^r both having t_H valid signatures.[25] For example, in a Coin-Fixed-To-0 step s, all verifiers in $HSV_1^{r,s}$ signed for bit 0 and $H(B^r)$, and propagated their messages. All verifiers in $MSV_1^{r,s}$ also signed 0 and $H(B^r)$, but withheld their messages. Because $|HSV_1^{r,s}| + |MSV_1^{r,s}| \geq t_H$, the system has enough signatures to certify B^r. However, since the malicious verifiers withheld their signatures, the users enter step $s+1$, which is a Coin-Fixed-To-1 step. Because $|HSV_1^{r,s}| < t_H$ due to the partition, the verifiers in $HSV_1^{r,s+1}$ did not see t_H signatures for bit 0 and they all signed for bit 1. All verifiers in $MSV_1^{r,s+1}$ did the same. Because $|HSV_1^{r,s+1}| + |MSV_1^{r,s+1}| \geq t_H$, the system has enough signatures to certify B_ϵ^r. The Adversary then creates a fork by releasing the signatures of $MSV_1^{r,s}$ for 0 and $H(B^r)$.

Accordingly, there will be two Q^r's, defined by the corresponding blocks of round r. However, the fork will not continue and only one of the two branches may grow in round $r+1$.

Additional Instructions for *Algorand'*$_2$. *When seeing a non-empty block B^r and the empty block B_ϵ^r, follow the non-empty one (and the Q^r defined by it).*

Indeed, by instructing the users to go with the non-empty block in the protocol, if a large amount of honest users in PK^{r+1-k} realize there is a fork at the beginning of round $r+1$, then the empty block will not have enough followers and will not grow. Assume the Adversary manages to partition the honest users so that some honest users see B^r (and perhaps B_ϵ^r), and some only see B_ϵ^r. Because the Adversary cannot tell which one of them will be a verifier following B^r and which will be a verifier following B_ϵ^r, the honest users are randomly partitioned and each one of them still becomes a verifier (either with respect to B^r or with respect to B_ϵ^r) in a step $s > 1$ with probability p. For the malicious users, each one of them may have two chances to become a verifier, one with B^r and the other with B_ϵ^r, each with probability p independently.

Let $HSV_{1;B^r}^{r+1,s}$ be the set of honest verifiers in step s of round $r+1$ following B^r. Other notations such as $HSV_{1;B_\epsilon^r}^{r+1,s}$, $MSV_{1;B^r}^{r+1,s}$ and $MSV_{1;B_\epsilon^r}^{r+1,s}$ are similarly defined. By Chernoff bound, it is easy

[25] Having a fork with two non-empty blocks is not possible with or without partitions, except with negligible probability.

to see that with overwhelming probability,

$$|HSV^{r+1,s}_{1;B^r}| + |HSV^{r+1,s}_{1;B^r_\epsilon}| + |MSV^{r+1,s}_{1;B^r}| + |MSV^{r+1,s}_{1;B^r_\epsilon}| < 2t_H.$$

Accordingly, the two branches cannot both have t_H proper signatures certifying a block for round $r+1$ in the same step s. Moreover, since the selection probabilities for two steps s and s' are the same and the selections are independent, also with overwhelming probability

$$|HSV^{r+1,s}_{1;B^r}| + |MSV^{r+1,s}_{1;B^r}| + |HSV^{r+1,s'}_{1;B^r_\epsilon}| + |MSV^{r+1,s'}_{1;B^r_\epsilon}| < 2t_H,$$

for any two steps s and s'. When $F = 10^{-18}$, by the union bound, as long as the Adversary cannot partition the honest users for a long time (say 10^4 steps, which is more than 55 hours with $\lambda = 10$ seconds[26]), with high probability (say $1 - 10^{-10}$) at most one branch will have t_H proper signatures to certify a block in round $r+1$.

Finally, if the physical partition has created two parts with roughly the same size, then the probability that $|HSV^{r,s}_i| + |MSV^{r,s}_i| \geq t_H$ is small for each part i. Following a similar analysis, even if the Adversary manages to create a fork with some non-negligible probability in each part for round r, at most one of the four branches may grow in round $r+1$.

10.2 Adversarial Partition

Second of all, the partition may be caused by the Adversary, so that the messages propagated by the honest users in one part will not reach the honest users in the other part directly, but the Adversary is able to forward messages between the two parts. Still, once a message from one part reaches an honest user in the other part, it will be propagated in the latter as usual. If the Adversary is willing to spend a lot of money, it is conceivable that he may be able to hack the Internet and partition it like this for a while.

The analysis is similar to that for the larger part in the physical partition above (the smaller part can be considered as having population 0): the Adversary may be able to create a fork and each honest user only sees one of the branches, but at most one branch may grow.

10.3 Network Partitions in Sum

Although network partitions can happen and a fork in one round may occur under partitions, there is no lingering ambiguity: a fork is very short-lived, and in fact lasts for at most a single round. In all parts of the partition except for at most one, the users cannot generate a new block and thus (a) realize there is a partition in the network and (b) never rely on blocks that will "vanish".

Acknowledgements

We would like to first acknowledge Sergey Gorbunov, coauthor of the cited Democoin system.

Most sincere thanks go to Maurice Herlihy, for many enlightening discussions, for pointing out that pipelining will improve Algorand's throughput performance, and for greatly improving the

[26]Note that a user finishes a step s without waiting for 2λ time only if he has seen at least t_H signatures for the same message. When there are not enough signatures, each step will last for 2λ time.

exposition of an earlier version of this paper. Many thanks to Sergio Rajsbaum, for his comments on an earlier version of this paper. Many thanks to Vinod Vaikuntanathan, for several deep discussions and insights. Many thanks to Yossi Gilad, Rotem Hamo, Georgios Vlachos, and Nickolai Zeldovich for starting to test these ideas, and for many helpful comments and discussions.

Silvio Micali would like to personally thank Ron Rivest for innumerable discussions and guidance in cryptographic research over more than 3 decades, for coauthoring the cited micropayment system that has inspired one of the verifier selection mechanisms of Algorand.

We hope to bring this technology to the next level. Meanwhile the travel and companionship are great fun, for which we are very grateful.

References

[1] *Bitcoin Computation Waste*, http://gizmodo.com/the-worlds-most-powerful-computer-network-is-being-was-50 2013.

[2] Bitcoinwiki. *Proof of Stake.* `http://www.blockchaintechnologies.com/blockchain-applications` As of 5 June 2016.

[3] Coindesk.com. *Bitcoin: A Peer-to-Peer Electronic Cash System* `http://www.coindesk.com/ibm-reveals-proof-concept-blockchain-powered-internet-things/` As of June 2016.

[4] Ethereum. *Ethereum.* `https://github.com/ethereum/`. As of 12 June 2016.

[5] HowStuffWorks.com. *How much actual money is there in the world?*, `https://money.howstuffworks.com/how-much-money-is-in-the-world.htm`. As of 5 June 2016.

[6] en.wikipedia.org/wiki/Sortition.

[7] M. Ben-Or. *Another advantage of free choice: Completely asynchronous agreement protocols.* Proc. 2nd Annual Symposium on Principles of Distributed Computing, ACM, New York, 1983, pp. 27-30.

[8] M. Castro and B. Liskov. *Practical Byzantine Fault Tolerance*, Proceedings of the Third Symposium on Operating Systems Design and Implementation. New Orleans, Louisiana, USA, 1999, pp. 173–186.

[9] D. L. Chaum, *Random Sample Elections*, `https://www.scribd.com/mobile/document/236881043/Random-Sa`

[10] B. Chor and C. Dwork. *Randomization in Byzantine agreement, in Randomness and Computation.* S. Micali, ed., JAI Press, Greenwich, CT, 1989, pp. 433-498.

[11] C. Decker and R. Wattenhofer. *Information Propagation in the Bitcoin Network.* 13-th IEEE Conference on Peer-to-Peer Computing, 2013.

[12] D. Dolev. *The Byzantine Generals Strike Again.* J. Algorithms, 3, (1982), pp. 14-30.

[13] D. Dolev and H.R. Strong. *Authenticated algorithms for Byzantine agreement.* SIAM Journal on Computing 12 (4), 656-666.

[14] C. Dwork and M. Naor. *Pricing via Processing, Or, Combatting Junk Mail.* Advances in Cryptology, CRYPTO'92: Lecture Notes in Computer Science No. 740. Springer: 139–147.

[15] P. Feldman and S. Micali. *An Optimal Probabilistic Algorithm for Synchronous Byzantine Agreement.* (Preliminary version in STOC 88.) SIAM J. on Computing, 1997.

[16] M. Fischer. *The consensus problem in unreliable distributed systems (a brief survey).* Proc. International Conference on Foundations of Computation, 1983.

[17] S. Goldwasser, S. Micali, and R. Rivest. *A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attack.* SIAM Journal of Computing, 17, No. 2, April 1988, pp. 281-308

[18] S. Gorbunov and S. Micali. *Democoin: A Publicly Verifiable and Jointly Serviced Cryptocurrency.* `https://eprint.iacr.org/2015/521`, May 30, 2015.

[19] J. Katz and C-Y Koo. *On Expected Constant-Round Protocols for Byzantine Agreement.* `https://www.cs.umd.edu/~jkatz/papers/BA.pdf`.

[20] A. Kiayias, A. Russel, B. David, and R. Oliynycov.. *Ouroburos: A provably secure proof-of-stake protocol.* Cryptology ePrint Archive, Report 2016/889, 2016. http://eprint.iacr.org/2016/889.

[21] S. King and S. Nadal. *PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake*, 2012.

[22] D. Lazar and Y. Gilad. Personal Communication.

[23] N. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.

[24] S. Micali. *Algorand: The Efficient Public Ledger.* `https://arxiv.org/abs/1607.01341`.

[25] S. Micali. *Fast And Furious Byzantine Agreement.* Innovation in Theoretical Computer Science 2017. Berkeley, CA, January 2017. Single-page abstract.

[26] S. Micali. *Byzantine Agreement, Made Trivial.* `https://people.csail.mit.edu/silvio/SelectedScientif`

[27] S. Micali, M. Rabin and S. Vadhan. *Verifiable Random Functions.* 40th Foundations of Computer Science (FOCS), New York, Oct 1999.

[28] S. Micali and R. L. Rivest. *Micropayments Revisited.* Lecture Notes in Computer Science, Vol. 2271, pp 149-163, Springer Verlag, 2002.

[29] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System.* `http://www.bitcoin.org/bitcoin.pdf`, May 2009.

[30] R. Pass and E. Shi. *The Sleepy Model of Consensus.* Cryptology ePrint Archive, Feb 2017, Report 2017/918.

[31] M. Pease, R. Shostak, and L. Lamport. *Reaching agreement in the presence of faults.* J. Assoc. Comput. Mach., 27 (1980), pp. 228-234.

[32] M. Rabin. *Randomized Byzantine generals.* 24th Foundations of Computer Science (FOCS), IEEE Computer Society Press, Los Alamitos, CA, 1983, pp. 403-409.

[33] R. Turpin and B. Coan. *Extending binary Byzantine agreement to multivalued Byzantine agreement.* Inform. Process. Lett., 18 (1984), pp. 73-76.

Algorand: Scaling Byzantine Agreements for Cryptocurrencies

Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, Nickolai Zeldovich
MIT CSAIL

ABSTRACT

Algorand is a new cryptocurrency that confirms transactions with latency on the order of a minute while scaling to many users. Algorand ensures that users never have divergent views of confirmed transactions, even if some of the users are malicious and the network is temporarily 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 125× 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 [24, 50] and fair protocols [2], 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 [46].

Double-spending is the core problem faced by any cryptocurrency, where an adversary holding $1 gives his $1 to two different users. Cryptocurrencies prevent double-spending by reaching consensus on an ordered log ("blockchain") of transactions. Reaching consensus is difficult because of the open setting: since anyone can participate, an adversary can create an arbitrary number of pseudonyms ("Sybils") [21], making it infeasible to rely on traditional consensus protocols [15] that require a fraction of honest users.

Bitcoin [42] and other cryptocurrencies [23, 54] address this problem using proof-of-work (PoW), where users must repeatedly compute hashes to grow the blockchain, 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 blockchains 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 (e.g., 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 designed to confirm transactions on the order of one minute. The core of Algorand uses a Byzantine agreement protocol called *BA*⋆ 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*⋆ suitable for Algorand is the use of verifiable random functions (VRFs) [39] to randomly select users in a private and non-interactive way. *BA*⋆ was previously presented at a workshop at a high level [38], and a technical report by Chen and Micali [16] described an earlier version of Algorand.

Algorand faces three challenges. First, Algorand must avoid Sybil attacks, where an adversary creates many pseudonyms to influence the Byzantine agreement protocol. Second, *BA*⋆ 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 re-

SOSP'17, October 28–31, 2017, Shanghai, China.

ACM ISBN 978-1-4503-5085-3/17/10.
https://doi.org/10.1145/3132747.3132757

silient to denial-of-service attacks, and continue to operate even if an adversary disconnects some of the users [30, 52].

Algorand addresses these challenges using several techniques, as follows.

Weighted users. To prevent Sybil attacks, Algorand assigns a weight to each user. *BA*⋆ 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 (over 2/3) of the money is owned by honest users, Algorand can avoid forks and double-spending.

Consensus by committee. *BA*⋆ achieves scalability by choosing a committee—a small set of representatives randomly 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 [39]) of their private key and public information from the blockchain. If the function indicates that the user is chosen, it returns a short string that proves this user's committee membership to other users, which the user can include in his network messages. 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 ~22 seconds with 50,000 users, that Algorand's latency remains nearly constant when scaling to half a million users, that Algorand achieves 125× 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 [42], 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 [42], 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 blockchain 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 [29].

By relying on Byzantine agreement, Algorand eliminates the possibility of forks, and avoids the need to reason about mining strategies [8, 25, 47]. As a result, transactions are confirmed on the order of a minute. To make the Byzantine agreement robust to Sybil attacks, Algorand associates weights with users according to the money they hold. Other techniques have been proposed in the past to resist Sybil attacks in Byzantine-agreement-based cryptocurrencies, including having participants submit security deposits and punishing those who deviate from the protocol [13].

Byzantine consensus. Byzantine agreement protocols have been used to replicate a service across a small group of servers, such as in PBFT [15]. Follow-on work has shown how to make Byzantine fault tolerance perform well and scale to dozens of servers [1, 17, 34]. 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 [36] 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 blockchains, which Algorand avoids.

Honey Badger [40] 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 5 minutes and achieve a throughput of 200 KBytes/sec of data appended to the ledger using 10 MByte blocks and 104 participating 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 better performance (confirming transactions in about a minute, reaching similar throughput) without having to choose a fixed set of servers ahead of time.

Bitcoin-NG [26] suggests using the Nakamoto consensus to elect a leader, and then have that leader publish blocks of transactions, resulting in an order of magnitude of improvement in latency of confirming transactions over Bitcoin. Hybrid consensus [31, 33, 43] refines the approach of using the Nakamoto consensus to periodically select a group of participants (e.g., every day), and runs a Byzantine agreement between selected participants to confirm transactions until new servers are selected. This allows improving performance over standard Nakamoto consensus (e.g., Bitcoin); for example, ByzCoin [33] provides a latency of about 35 seconds and a throughput of 230 KBytes/sec of data appended to the ledger with an 8 MByte block size and 1000 participants in the Byzantine agreement. Although Hybrid consensus makes the set of Byzantine servers dynamic, it opens up the possibility of forks, 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 [43] acknowledges that the Hybrid consensus design is secure only with respect to a "mildly adaptive" adversary that cannot compromise the selected servers within a day (the participant selection interval), and explicitly calls out the open problem of handling fully adaptive adversaries. 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 [37] takes an alternative approach to using Byzantine consensus in a cryptocurrency, where each user can trust quorums of other users, forming a trust hierarchy. Consistency is ensured as long as all transactions share at least one transitively trusted quorum of users, and sufficiently many of these users are 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, suggested as an alternative or enhancement to proof-of-work [3, 10]. There is a key difference, however, between Algorand using monetary value as weights and many proof-of-stake cryptocurrencies. In many proof-of-stake cryptocurrencies, 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. 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 "detect and punish" malicious users, but this is not required to prevent forks or double spending.

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 [4]. Since no work is involved in generating blocks, a malicious leader can announce one block, and then present some other block to isolated users. Attackers may also split their credits among several "users", who might get selected as leaders, to minimize the penalty when a bad leader is caught. Therefore some proof-of-stake cryptocurrencies require a master key to periodically sign the correct branch of the ledger in order to mitigate forks [32]. This raises significant trust concerns regarding the currency, and has also caused accidental forks in the past [44]. Algorand answers this challenge by avoiding forks, even if the leader turns out to be malicious.

Ouroboros [31] is a recent proposal for realizing proof-of-stake. For security, Ouroboros assumes that honest users can communicate within some bounded delay (i.e., a strongly synchronous network). Furthermore, it selects some users to participate in a joint-coin-flipping protocol and assumes that most of them are incorruptible by the adversary for a significant epoch (such as a day). In contrast Algorand assumes that the adversary may temporarily fully control the network and immediately corrupt users in targeted attacks.

Trees and DAGs instead of chains. GHOST [48], SPECTRE [49], and Meshcash [5] are recent proposals for increasing Bitcoin's throughput by replacing the underlying chain-structured ledger with a tree or directed acyclic graph (DAG) structures, and resolving conflicts in the forks of these data structures. These protocols rely on the Nakamoto consensus using proof-of-work. By carefully designing the selection rule between branches of the trees/DAGs, they are able to substantially increase the throughput. In contrast, Algorand is focused on eliminating forks; 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 allows users to agree on an ordered log of transactions, and achieves two goals with respect to the log:

Safety goal. With overwhelming probability, all users agree on the same transactions. More precisely, if one honest

user accepts transaction A (i.e., it appears in the log), then any future transactions accepted by other honest users will appear in a log that already contains A. This holds even for isolated users that are disconnected from the network—e.g., by Eclipse attacks [29].

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

Assumptions. 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 h (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 achieve liveness, Algorand makes a "strong synchrony" assumption that most honest users (e.g., 95%) can send messages that will be received by most other honest users (e.g., 95%) within a known time bound. This assumption allows the adversary to control the network of a few honest users, but does not allow the adversary to manipulate the network at a large scale, and does not allow network partitions.

Algorand achieves safety with a "weak synchrony" assumption: the network can be asynchronous (i.e., entirely controlled by the adversary) for a long but bounded period of time (e.g., at most 1 day or 1 week). After an asynchrony period, the network must be strongly synchronous for a reasonably long period again (e.g., a few hours or a day) for Algorand to ensure safety. More formally, the weak synchrony assumption is that in every period of length b (think of b as a day or a week), there must be a strongly synchronous period of length $s < b$ (an s of a few hours suffices).

Finally, Algorand assumes loosely synchronized clocks across all users (e.g., using NTP) in order to recover liveness after weak synchrony. Specifically, the clocks must be close enough in order for most honest users to kick off the recovery protocol at approximately the same time. If the clocks are out of sync, the recovery protocol does not succeed.

4 OVERVIEW

Algorand requires each user to have a public key. 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 and a pointer to the previous block, is appended to the blockchain. In the rest of this paper, we refer to Algorand software running on a user's computer as that user.

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 $BA\star$ to reach consensus on one of these pending blocks.

$BA\star$ executes in *steps*, communicates over the same gossip protocol, and produces a new agreed-upon block. $BA\star$ can produce two kinds of consensus: *final* consensus and *tentative* consensus. If one user reaches final consensus, this means that any other user that reaches final or tentative consensus in the same round must agree on the same block value (regardless of whether the strong synchrony assumption held). This ensures Algorand's safety, since this means that all future transactions will be chained to this final block (and, transitively, to its predecessors). Thus, Algorand confirms a transaction when the transaction's block (or any successor block) reaches final consensus. On the other hand, tentative consensus means that other users may have reached tentative consensus on a different block (as long as no user reached final consensus). A user will confirm a transaction from a tentative block only if and when a successor block reaches final consensus.

$BA\star$ produces tentative consensus in two cases. First, if the network is strongly synchronous, an adversary may, with small probability, be able to cause $BA\star$ to reach tentative consensus on a block. In this case, $BA\star$ will not reach consensus on two different blocks, but is simply unable to confirm that the network was strongly synchronous. Algorand will eventually (in a few rounds) reach final consensus on a successor block, with overwhelming probability, and thus confirm these earlier transactions.

The second case is that the network was only weakly synchronous (i.e., it was entirely controlled by the adversary, with an upper bound on how long the adversary can keep control). In this case, $BA\star$ can reach tentative consensus on two different blocks, forming multiple forks. This can in turn prevent $BA\star$ from reaching consensus again, because the users are split into different groups that disagree about previous blocks. To recover liveness, Algorand periodically invokes $BA\star$ to reach consensus on *which fork* should be used going forward. Once the network regains strong synchrony, this will allow Algorand to choose one of the forks, and then reach final consensus on a subsequent block on that fork.

We now describe how Algorand's components fit together.

Gossip protocol. Algorand implements a gossip network (similar to Bitcoin) 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 forwarding loops, users do not

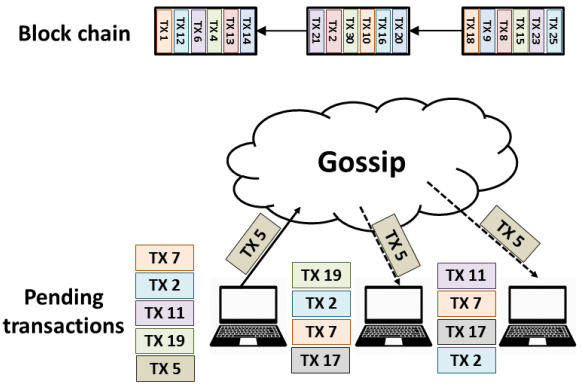

Figure 1: An overview of transaction flow in Algorand.

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 if 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. Since sortition is random, there may be multiple users selected to propose a block, and the priority determines which block everyone should adopt. 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, block proposals 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 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 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. (Steps are not synchronized across users; each user checks for selection as soon as he observes the previous step had ended.) 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.

Efficiency. When the network is strongly synchronous, *BA*⋆ guarantees that if all honest users start with the same initial block (i.e., the highest priority block proposer was honest), then *BA*⋆ establishes final consensus over that block

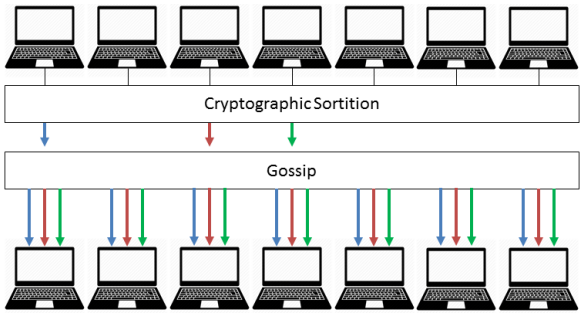

Figure 2: An overview of one step of *BA*⋆. To simplify the figure, each user is shown twice: once at the top of the diagram and once at the bottom. Each arrow color indicates a message from a particular user.

and terminates precisely in 4 interactive steps between users. Under the same network conditions, and in the worst case of a particularly lucky adversary, all honest users reach consensus on the next block within expected 13 steps, as analyzed in Appendix C of the technical report [27].

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 $W = \sum_i w_i$, 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 i to have a public/private key pair, (pk_i, sk_i).

Sortition is implemented using verifiable random functions (VRFs) [39]. Informally, on any input string x, $VRF_{sk}(x)$ returns two values: a hash and a proof. The hash is a *hashlen*-bit-long value that is uniquely determined by sk and x, but is indistinguishable from random to anyone that does not know sk. The proof π enables anyone that knows pk to check that the hash indeed corresponds to x, without having to know sk. For security, we require that the VRF provides these properties even if pk and sk are chosen by an attacker.

5.1 Selection procedure

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 member of the committee at a certain step of *BA*⋆. Algorand specifies a threshold τ that determines the expected number of users selected for that role.

It is important that sortition selects users in proportion to their weight; otherwise, sortition would not defend against Sybil attacks. One subtle implication is that users may be chosen more than once by sortition (e.g., because they have a high weight). Sortition addresses this by returning the j parameter, which indicates how many times the user was

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.

chosen. Being chosen j times means that the user gets to participate as j different "sub-users."

To select users in proportion to their money, we consider each unit of Algorand as a different "sub-user." If user i owns w_i (integral) units of Algorand, then simulated user (i,j) with $j \in \{1,\ldots,w_i\}$ represents the j^{th} unit of currency i owns, and is selected with probability $p = \frac{\tau}{W}$, where W is the total amount of currency units in Algorand.

As shown in Algorithm 1, a user performs sortition by computing $\langle hash, \pi \rangle \leftarrow VRF_{sk}(seed||role)$, where sk is the user's secret key. The pseudo-random $hash$ determines how many sub-users are selected, as follows. The probability that exactly k out of the w (the user's weight) sub-users are selected follows the binomial distribution, $B(k;w,p) = \binom{w}{k} p^k (1-p)^{w-k}$, where $\sum_{k=0}^{w} B(k;w,p) = 1$. Since $B(k_1;n_1,p) + B(k_2;n_2,p) = B(k_1+k_2;n_1+n_2,p)$, splitting a user's weight (currency) among Sybils does not affect the number of selected sub-users under his/her control.

To determine how many of a user's w sub-users are selected, the sortition algorithm divides the interval $[0,1)$ into consecutive intervals of the form $I^j = \left[\sum_{k=0}^{j} B(k;w,p), \sum_{k=0}^{j+1} B(k;w,p)\right)$ for $j \in \{0,1,\ldots,w\}$. If $hash/2^{hashlen}$ (where $hashlen$ is the bit-length of $hash$) falls in the interval I^j, then the user has exactly j selected sub-users. The number of selected sub-users is publicly verifiable using the proof π (from the VRF output).

Sortition provides two important properties. First, given a random seed, the VRF outputs a pseudo-random hash value, which is essentially uniformly distributed between 0 and $2^{hashlen} - 1$. As a result, users are selected at random based on their weights. Second, an adversary that does not know sk_i cannot guess how many times user i is chosen, or if i was chosen at all (more precisely, the adversary cannot guess any better than just by randomly guessing based on the weights).

The pseudocode for verifying a sortition proof, shown in Algorithm 2, follows the same structure to check if that user was selected (the weight of the user's public key is obtained from the ledger). The function returns the number of selected sub-users (or zero if the user was not selected at all).

5.2 Choosing the seed

Sortition requires a seed that is chosen at random and publicly known. For Algorand, each round requires a 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 selection of corrupted users.

In each round of Algorand a new seed is published. The seed published at 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 the seed for that round being determined (§5.3). 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 included invalid transactions), 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 (after their public keys are declared) using distributed random number generation [14].

To limit the adversary's ability to manipulate sortition, and thus manipulate the selection of users for different committees, the selection seed (passed to Algorithm 1 and Algorithm 2) is refreshed once every R rounds. Namely, at round r Algorand calls the sortition functions with $seed_{r-1-(r \bmod R)}$.

5.3 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 the selection seed used in that round, i.e., $seed_{r-1-(r \bmod R)}$. When the network is not strongly synchronous, the attacker has complete control over the links, and can therefore drop block proposals and force users to agree on empty blocks, such that future selection seeds can be computed. To mitigate such attacks Algorand relies on the weak synchrony assumption (in every period of length b, there must be a strongly synchronous period of length $s < b$). Whenever Algorand performs cryptographic sortition for round r, it checks the timestamp included in the agreed-upon block for round $r-1-(r \bmod R)$, and uses the keys (and associated weights) from the last block that was created b-time before block $r-1-(r \bmod R)$. The lower

bound s on the length of a strongly synchronous period should allow for sufficiently many blocks to be created in order to ensure with overwhelming probability that at least one block was honest. This ensures that, as long as s is suitably large, an adversary u choosing a key sk_u cannot predict the seed for round r.

This look-back period b has the following trade-off: a large b mitigates the risk that attackers are able break the weak synchronicity assumption, yet it increases the chance that users have transferred their currency to someone else and therefore have nothing to lose if the system's security breaks. This is colloquially known as the "nothing at stake" problem; one possible way to avoid this trade-off, which we do not explore in Algorand, is to take the minimum of a user's current balance and the user's balance from the look-back block as the user's weight.

Appendix A of the technical report [27] formally analyzes the number of blocks that Algorand needs to be created in the period s when the network is strongly connected. We show that to ensure a small probability of failure F, the number of blocks is logarithmic in $\frac{1}{F}$, which allows us to obtain high security with a reasonably low number of required blocks.

6 BLOCK PROPOSAL

To ensure that some block is proposed in each round, Algorand sets the sortition threshold for the block-proposal role, τ_{PROPOSER}, to be greater than 1 (although Algorand will reach consensus on at most one of these proposed blocks). Appendix B of the technical report [27] proves that choosing $\tau_{\text{PROPOSER}} = 26$ 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 (e.g., $1 - 10^{-11}$).

Minimizing unnecessary block transmissions. One risk of choosing several 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: For each selected sub-user $1, \ldots, j$ of user i, the priority for the block proposal is obtained by hashing the (verifiably random) *hash* output of VRF concatenated with the sub-user index. The highest priority of all the block proposer's selected sub-users is the priority of the block.

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 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 quickly discard other proposed blocks.

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 completed round $r-1$, sufficiently many users 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, 5 seconds each. 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 $1-h$, by Algorand's assumption that at least $h > 2/3$ 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 one of two options. In the second phase, $BA\star$ reaches agreement on one of these options: either agreeing on a proposed block, or agreeing on an empty block.

Each phase consists of several interactive *steps*; the first phase always takes two steps, and the second phase takes two steps if the highest-priority block proposer was honest (sent the same block to all users), and as we show in our analysis 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.

In each step, every committee member casts a vote for some value, and all users count the votes. Users that receive more than a threshold of votes for some value will vote for that value in the next step (if selected as a committee member). If the users do not receive enough votes for any value, they time out, and their choice of vote for the next step is determined by the step number.

In the common case, when the network is strongly synchronous and the highest-priority block proposer was honest, *BA*⋆ reaches *final* consensus by using its final step to confirm that there cannot be any other agreed-upon block in the same round. Otherwise, *BA*⋆ may declare *tentative* consensus if it cannot confirm the absence of other blocks due to possible network asynchrony.

A key aspect of *BA*⋆'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: verify signatures, count votes, and reach the agreement decision.

7.1 Main procedure of *BA*⋆

The top-level procedure implementing *BA*⋆, as invoked by Algorand, is shown in Algorithm 3. The procedure takes a context *ctx*, which captures the current state of the ledger, a *round* number, and a new proposed *block*, from the highest-priority block proposer (§6). Algorand is responsible for ensuring that the block is valid (by checking the proposed block's contents and using an empty block if it is invalid, as described in §8). The context consists of the seed for sortition, the user weights, and the last agreed-upon block.

For efficiency, *BA*⋆ votes for hashes of blocks, instead of entire block contents. At the end of the *BA*⋆ algorithm, we use the BlockOfHash() function to indicate that, if *BA*⋆ has not yet received the pre-image of the agreed-upon hash, it must obtain it from other users (and, since the block was agreed upon, many of the honest users must have received it during block proposal).

The *BA*⋆ algorithm also determines whether it established final or tentative consensus. We will discuss this check in detail when we discuss Algorithm 8.

7.2 Voting

Sending votes (Algorithm 4). Algorithm 4 shows the pseudocode for CommitteeVote(), which checks if the user is selected for the committee in a given *round* and *step* of *BA*⋆. The CommitteeVote() procedure invokes Sortition() from Algorithm 1 to check if the user is chosen to participate in the committee. If the user is chosen for this step, the user gossips a signed message containing the value passed to CommitteeVote(), which is typically the hash of some block.

```
procedure BA⋆(ctx, round, block):
hblock ← Reduction(ctx, round, H(block))
hblock⋆ ← BinaryBA⋆(ctx, round, hblock)
// Check if we reached "final" or "tentative" consensus
r ← CountVotes(ctx, round, FINAL, T_FINAL, τ_FINAL, λ_STEP)
if hblock⋆ = r then
    return ⟨FINAL, BlockOfHash(hblock⋆)⟩
else
    return ⟨TENTATIVE, BlockOfHash(hblock⋆)⟩
```

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

```
procedure CommitteeVote(ctx, round, step, τ, value):
// check if user is in committee using Sortition (Alg. 1)
role ← ⟨"committee", round, step⟩
⟨sorthash, π, j⟩ ← Sortition(user.sk, ctx.seed, τ, role,
        ctx.weight[user.pk], ctx.W)
// only committee members originate a message
if j > 0 then
    Gossip(⟨user.pk, Signed_user.sk(round, step,
            sorthash, π, H(ctx.last_block), value)⟩)
```

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

To bind the vote to the context, the signed message includes the hash of the previous block.

Counting votes (Algorithm 5 and Algorithm 6). The CountVotes() procedure (Algorithm 5) reads messages that belong to the current round and step from the *incomingMsgs* buffer. (For simplicity, our pseudocode assumes that a background procedure takes incoming votes and stores them into that buffer, indexed by the messages' round and step.) It processes the votes by calling the ProcessMsg() procedure for every message (Algorithm 6), which ensures that the vote is valid. Note that no private state is required to process these messages.

ProcessMsg() returns not just the value contained in the message, but also the number of votes associated with that value. If the message was not from a chosen committee member, ProcessMsg() returns zero votes. If the committee member was chosen several times (see §5), the number of votes returned by ProcessMsg() reflects that as well. ProcessMsg() also returns the sortition hash, which we will use later in Algorithm 9.

As soon as one value has more than $T \cdot \tau$ votes, CountVotes() returns that value. τ is the expected number of users that Sortition() selects for the committee, and is the same for each step (τ_{STEP}) with the exception of the final step (τ_{FINAL}). T is a fraction of that expected committee size ($T > \frac{2}{3}$) that defines *BA*⋆'s voting threshold; this is also the same for every step except the final step, and we analyze it in §7.5. If not enough messages were received within the allocated λ time window, then CountVotes() produces TIMEOUT.

```
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*.

The threshold ensures that if one honest user's CountVotes() returns a particular value, then all other honest users will return either the same value or TIMEOUT, even under the weak synchrony assumption (see Lemma 1 in Appendix C.2 of the technical report [27]).

7.3 Reduction

The Reduction() procedure, shown in Algorithm 7, converts the problem of reaching consensus on an arbitrary value (the hash of a block) to reaching consensus on one of two values: either a specific proposed block hash, or the hash of an empty block. Our reduction is inspired by Turpin and Coan's two-step technique [51]. This reduction is important to ensure liveness.

In the first step of the reduction, each committee member votes for the hash of the block passed to Reduction() by $BA\star()$. In the second step, committee members vote for the hash that received at least $T \cdot \tau$ votes in the first step, or the hash of the default empty block if no hash received enough votes. Reduction() ensures that there is at most one non-empty block that can be returned by Reduction() for all honest users.

```
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.

In the common case when the network is strongly synchronous and the highest-priority block proposer was honest, most (e.g., 95%) of the users will call Reduction() with the same *hblock* parameter, and Reduction() will return that same *hblock* result to most users as well.

On the other hand, if the highest-priority block proposer was dishonest, different users may start Reduction() with different *hblock* parameters. In this case, no single *hblock* value may be popular enough to cross the threshold of votes. As a result, Reduction() will return *empty_hash*.

7.4 Binary agreement

Algorithm 8 shows Binary$BA\star()$, which reaches consensus on one of two values: either the hash passed to Binary$BA\star()$ or the hash of the empty block. Binary$BA\star()$ relies on Reduction() to ensure that at most one non-empty block hash is passed to Binary$BA\star()$ by all honest users.

Safety with strong synchrony. In each step of Binary$BA\star()$, a user who has seen more than $T \cdot \tau$ votes for some value will vote for that same value in the next step (if selected). However, if no value receives enough votes, Binary$BA\star()$ chooses the next vote in a way that ensures consensus in a strongly synchronous network.

Specifically, user A may receive votes from an adversary that push the votes observed by A past the threshold, but the adversary might not send the same votes to other users (or might not send them in time). As a result, A returns consensus on a block, but other users timed out in that step. It is crucial that Binary$BA\star()$ chooses the votes for the next step in a way that will match the block already returned by A. Algorithm 8 follows this rule: every **return** statement

```
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*.

is coupled with a check for TIMEOUT that sets the next-step vote to the same value that could have been returned.

It is also crucial that Binary*BA*⋆() is able to collect enough votes in the next step to carry forward the value that A already reached consensus on. If there are many users like A that have already returned consensus, Binary*BA*⋆() may not have enough users to push CountVotes() in the next step past the threshold. To avoid this problem, whenever a user returns consensus, that user votes in the next three steps with the value they reached consensus on.

In the common case, when the network is strongly synchronous and the block proposer was honest, Binary*BA*⋆() will start with the same *block_hash* for most users, and will reach consensus in the first step, since most committee members vote for the same *block_hash* value.

Safety with weak synchrony. If the network is not strongly synchronous (e.g., there is a partition), Binary*BA*⋆() may return consensus on two different blocks. For example, suppose that, in the first step of Binary*BA*⋆(), all users vote for *block_hash*, but only one honest user, A, receives those votes. In this case, A will return consensus on *block_hash*, but all other users will move on to the next step. Now, the other users vote for *block_hash* again, because CountVotes() returned TIMEOUT. However, let's assume the network drops all of these votes. Finally, the users vote for *empty_hash* in the third step, the network becomes well behaved, and all votes are delivered. As a result, the users will keep voting for *empty_hash* until the next iteration of the loop, at which point they reach consensus on *empty_hash*. This is undesirable because Binary*BA*⋆() returned consensus on two different hashes to different honest users.

BA⋆() addresses this problem by introducing the notion of *final* and *tentative* consensus. Final consensus means that *BA*⋆() will not reach consensus on any other block for that round. Tentative consensus means that *BA*⋆() was unable to guarantee safety, either because of network asynchrony or due to a malicious block proposer.

BA⋆() designates consensus on value V as "final" if Binary*BA*⋆() reached consensus on V in the very first step, and if enough users observed this consensus being reached. Specifically, Binary*BA*⋆() sends out a vote for the special FINAL step to indicate that a user reached consensus on some value in the very first step, and *BA*⋆() collects these votes to determine whether final consensus was achieved. In a strongly synchronous network with an honest block proposer, Binary*BA*⋆() will reach consensus in the first step, most committee members will vote for the consensus block in the special FINAL step in Binary*BA*⋆(), and will receive more than a threshold of such votes in *BA*⋆(), thus declaring the block as final. The FINAL step is analogous to the final confirmation step implemented in many Byzantine-resilient protocols [15, 35].

Intuitively, this guarantees safety because a large threshold of users have already declared consensus for V, and will not vote for any other value in the same round. In our example above, where user A reached consensus on a different block than all other users, neither block would be designated as final, because only one user (namely, A) observed consensus at the first step, and there would never be enough votes to mark that block as final. Appendix C.1 of the technical report [27] formalizes and proves this safety property.

One subtle issue arises due to the fact that *BA*⋆ relies on a committee to declare final consensus, instead of relying on all participants. As a result, even if one user observes final consensus, an adversary that controls the network may be able to prevent a small fraction of other users from reaching any kind of consensus (final or tentative) for an arbitrary number of steps. Each of these steps give the adversary an additional small probability of reaching consensus on a different value (e.g., the empty block). To bound the total

```
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.

probability of an adversary doing so, *BA*⋆ limits the total number of allowed steps; Appendix C.1 of the technical report [27] relies on this. If the protocol runs for more than MaxSteps steps, *BA*⋆ halts without consensus and relies on the recovery protocol described in §8.2 to recover liveness.

Getting unstuck. One remaining issue is that consensus could get stuck if the honest users are split into two groups, A and B, and the users in the two groups vote for different values (say, we are in step 1, *A* votes for *empty_hash*, and *B* votes for *block_hash*). Neither group is large enough to gather enough votes on their own, but together with the adversary's votes, group A is large enough. In this situation, the adversary can determine what every user will vote for in the next step. To make some user vote for *empty_hash* in the next step, the adversary sends that user the adversary's own votes for *empty_hash* just before the timeout expires, which, together with *A*'s votes, crosses the threshold. To make the user vote for *block_hash*, the adversary does not send any votes to that user; as a result, that user's CountVotes() will return TIMEOUT, and the user will choose *block_hash* for the next step's vote, according to the Binary*BA*⋆() algorithm. This way, the adversary can split the users into two groups in the next step as well, and continue this attack indefinitely.

The attack described above requires the adversary to know how a user will vote after receiving TIMEOUT from CountVotes(). The third step of Binary*BA*⋆() is designed to avoid this attack by pushing towards accepting either *block_hash* or *empty_hash* based on a random "common coin," meaning a binary value that is predominantly the same for all users. Although this may sound circular, the users need not reach formal consensus on this common coin. As long as enough users observe the same coin bit, and the bit was not known to the attacker in advance of the step, Binary*BA*⋆() will reach consensus in the next iteration of the loop with probability 1/2 (i.e., the probability that the attacker guessed wrong). By repeating these steps, the probability of consensus quickly approaches 1.

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 9. If a user gets multiple votes (i.e., several of their sub-users were selected), then CommonCoin() considers multiple hashes from that user, by hashing that user's sortition hash with the sub-user index. Notice that hashes are random (since they are produced by hashing the pseudo-random VRF output), so their least-significant bits are also random. The common coin is used only when CountVotes() times out, giving sufficient time for all votes to propagate through the network. If the committee member with the lowest hash is honest, then all users that received his message observe the same coin.

If a malicious committee-member happens to hold the lowest 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 h (the fraction of money held by honest users), and thus there is at least an $h > \frac{2}{3}$ probability that the lowest sortition hash holder will be honest, which leads to consensus with probability $\frac{1}{2} \cdot h > \frac{1}{3}$ at each loop iteration. This allows Appendix C.3 of the technical report [27] to show that, with strong synchrony, *BA*⋆ does not exceed MaxSteps with overwhelming probability.

7.5 Committee size

The fraction $h > \frac{2}{3}$ of weighted honest users in Algorand must translate into a "sufficiently honest" committee for *BA*⋆. *BA*⋆ has two parameters at its disposal: τ, which controls the *expected* committee size, and T, which controls the number of votes needed to reach consensus ($T \cdot \tau$). We would like T to be as small as possible for liveness, but the smaller T is, the larger τ needs to be, to ensure that an adversary does not obtain enough votes by chance. Since a larger committee translates into a higher bandwidth cost, we choose two different parameter sets: T_{FINAL} and τ_{FINAL} for the FINAL step, which ensures an overwhelming probability of safety regardless of strong synchrony, and T_{STEP} and τ_{STEP} for all other steps, which achieve a reasonable trade-off between liveness, safety, and performance.

To make the constraints on τ_{STEP} and T_{STEP} precise, let us denote the number of honest committee members by g and the malicious ones by b; in expectation, $b + g = \tau_{\text{STEP}}$, but $b + g$ can vary since it is chosen by sortition. To ensure liveness, as we prove in Appendix C.2 of the technical report [27], *BA*⋆ requires $\frac{1}{2}g + b \leq T_{\text{STEP}} \cdot \tau_{\text{STEP}}$ and $g > T_{\text{STEP}} \cdot \tau_{\text{STEP}}$.

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. Figure 3 plots the expected committee size τ_{STEP} that is needed to satisfy both constraints, as a function of h, for a probability of violation of 5×10^{-9}; Appendix B of the technical report [27] describes this computation in more detail. The figure shows a trade-off: the weaker the assumption on the fraction of money held by honest users (h), the larger the committee size needs to be. The results show that, as h approaches $\frac{2}{3}$, the committee size grows quickly. However, at $h = 80\%$, $\tau_{\text{STEP}} = 2{,}000$ can ensure that these constraints hold with probability $1 - 5 \times 10^{-9}$ (using $T_{\text{STEP}} = 0.685$).

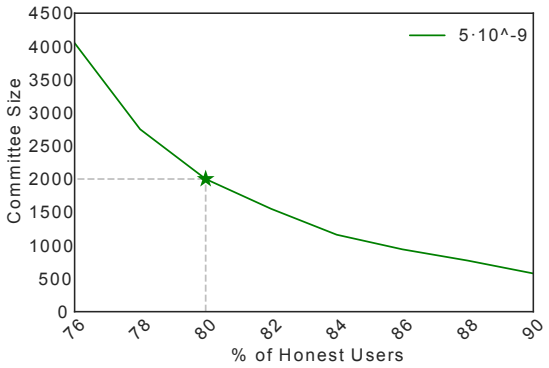

Figure 3: The committee size, τ, sufficient to limit the probability of violating safety to 5×10^{-9}. The x-axis specifies h, the weighted fraction of honest users. ⋆ marks the parameters selected in our implementation.

The constraints on τ_{FINAL} and T_{FINAL} are dictated by the proof of safety under weak synchrony; Appendix C.1 of the technical report [27] shows that $\tau_{\text{FINAL}} = 10,000$ suffices with $T_{\text{FINAL}} = 0.74$.

With these parameters, *BA*⋆ ensures safety even if the lowest-priority block proposer is malicious (proposes different blocks). Appendix C of the technical report [27] provides proofs of *BA*⋆'s safety under weak synchrony (§C.1), liveness under strong synchrony (§C.2), and efficiency (§C.3).

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.

8.1 Block format

Algorand's blocks consist of a list of transactions, along with metadata needed by *BA*⋆. Specifically, the metadata consists of the round number, the proposer's VRF-based seed (§6), a hash of the previous block in the ledger, and a timestamp indicating when the block was proposed. The list of transactions in a block logically translates to a set of weights for each user's public key (based on the balance of currency for that key), along with the total weight of all outstanding currency.

Once a user receives a block from the highest-priority proposer, the user validates the block contents before passing it on to *BA*⋆. In particular, the user checks that all transactions are valid; that the seed is valid; that the previous block hash is correct; that the block round number is correct; and that the timestamp is greater than that of the previous block and also approximately current (say, within an hour). If any of them are incorrect, the user passes an empty block to *BA*⋆.

8.2 Safety and liveness

To a large extent, Algorand relies on *BA*⋆ to reach consensus on blocks in the ledger. Algorand confirms transactions only when they appear in a final block, or in the predecessor of a final block. Final blocks guarantee that no other block could have reached consensus in the same round. This means that all final blocks are totally ordered with respect to one another, since (1) blocks form a linear chain, and (2) there can be exactly one final block at any given position in the chain. In other words, given two final blocks, one of them (the one with the smaller round number r_1) must be a predecessor of the other (the one with the higher round number r_2), since there must be *some* predecessor of the r_2 block in round r_1, and the safety condition guarantees that the r_1 block is the only possible such block.

The remaining issue is that, if the network is not strongly synchronous, *BA*⋆ may create forks (i.e., different users reach consensus on different blocks). This does not violate safety, because *BA*⋆ will return *tentative* consensus in this situation. However, forks do impact liveness: users on different forks will have different *ctx.last_block* values, which means they will not count each others' votes. As a result, at least one of the forks (and possibly all of the forks) will not have enough participants to cross the vote threshold, and *BA*⋆ will not be able to reach consensus on any more blocks on that fork.

To resolve these forks, Algorand periodically proposes a fork that all users should agree on, and uses *BA*⋆ to reach consensus on whether all users should, indeed, switch to this fork. To determine the set of possible forks, Algorand users passively monitor all *BA*⋆ votes (i.e., even votes whose *prev_hash* value does not match the current user's chain), and keep track of all forks. Users then use loosely synchronized clocks to stop regular block processing and kick off the recovery protocol at every time interval (e.g., every hour), which will propose one of these forks as the fork that everyone should agree on.

The recovery protocol starts by having users propose a fork using the block proposal mechanism (§6). Specifically, if a user is chosen to be a "fork proposer," that user proposes an empty block whose predecessor hash is the longest fork (by the number of blocks) observed by the user so far. Each user waits for the highest-priority fork proposal, much as in the block proposal mechanism. Each user validates the proposed block, by ensuring that the block's parent pointer is a chain that is as long as the longest chain seen by that user. Choosing the longest fork ensures that this fork will include all final blocks. Finally, the user invokes *BA*⋆ to reach consensus on this block, passing the round number found in the proposed block.

In order for *BA*⋆ to reach consensus on one of the forks, all Algorand users must use the same seed and user weights. This means that Algorand must use user weights and seeds from before any possible forks occurred. To do this, Algorand relies on the weak synchrony assumption—namely, that in every period of length b (think of b as 1 day), there must be a strongly synchronous period of length $s < b$ (think of s as a few hours). Under this assumption, using the block timestamps, Algorand quantizes time into b-long periods (think days), and finds the most recent block from the next-to-last complete b-long period. Algorand then uses the seed

from this block, and uses user weights from the last block that was agreed upon at least b-long time before it (§5.3).

Algorand takes the seed from the block from the *next-to-last* b-long period because the most recent b-long period may still have an unresolved fork. Such a fork would prevent users from agreeing on the seed and weights used in the recovery. However, as long as Algorand can recover within the s-long strongly synchronous period in the most recent b-long period, all users will agree on the same block from the next-to-last period (as long as their clocks are roughly synchronized).

To ensure that Algorand recovers from a fork (i.e., most honest users switch to the same fork) within the s-long synchronous period, Algorand users repeatedly attempt to reach consensus on a fork (applying a hash function to the seed each time to produce a different set of proposers and committee members), until they achieve consensus. Since, by assumption, Algorand is operating in a strongly synchronous period, it is not important whether $BA\star$ returns "final" or "tentative" consensus in this case. When Algorand is recovering outside of a strongly synchronous period, we cannot ensure recovery within s time.

8.3 Bootstrapping

Bootstrapping the system. To deploy Algorand, a common genesis block must be provided to all users, along with the initial cryptographic sortition seed. The value of $seed_0$ specified in the genesis block is decided using distributed random number generation [14], after the public keys and weights for the initial set of participants are publicly known.

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 $BA\star$ consensus outcomes. To help users catch up, Algorand generates a *certificate* for every block that was agreed upon by $BA\star$ (including empty blocks). The certificate is an aggregate of the votes from the last step of Binary$BA\star$() (not including the FINAL step) that would be sufficient to allow any user to reach the same conclusion by processing these votes (i.e., there must be at least $\lfloor T_{\text{STEP}} \cdot \tau_{\text{STEP}} \rfloor + 1$ votes). Importantly, the users must check the sortition hashes and proofs just like in Algorithm 6, and that all messages in the certificate are for the same Algorand round and $BA\star$ step.

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. Users can also request a certificate proving the safety of a block; this is simply the collection of votes for the FINAL step. Since final blocks are totally ordered, users need to check the safety of only the most recent block.

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 (up to MAXSTEPS) 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 $\tau_{\text{STEP}} > 1{,}000$, the probability of this attack is less than 2^{-166} at every step, making this attack infeasible.

Storage. The block history and matching certificates allow new users to catch-up, and are not required for users who are already up-to-date with the current ledger. Therefore Algorand distributes certificate and block storage across users. For N shards, users store blocks/certificates whose round number equals their public key modulo N.

8.4 Communication

Gossiping blocks and relaying messages. 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 [19] to make it difficult for an adversary to quickly disconnect a user.

To avoid an adversary from sending garbage messages and overwhelming Algorand's gossip network, Algorand nodes must validate messages before relaying them. Specifically, Algorand nodes should validate each message using Algorithm 6, and avoid relaying more than one message signed by a given public key per ⟨round, step⟩.

Scalability. The communication costs for each user depend on the expected size of the committee and the number of block proposers, which Algorand sets through τ_{PROPOSER}, τ_{STEP}, and τ_{FINAL} (independent of the number of users). As more users join, it takes a message longer to disseminate in the gossip network. Algorand's gossip network forms a random network graph (each user connects to random peers). Our theoretical analysis suggests that almost all users will be part of one connected component in the graph, and that dissemination time grows with the diameter of that component, which is logarithmic in the number of users [45]. Experi-

ments confirm that Algorand's performance is only slightly affected by more users (§10).

Since our random graph uses a fixed number of peers, one potential concern is that it may contain disconnected components [22]. However, only a small fraction of users might end up in a disconnected component, which does not pose a problem for $BA\star$. Moreover, Algorand replaces gossip peers each round, which helps users recover from being possibly disconnected in a previous round.

9 IMPLEMENTATION

We implemented a prototype of Algorand in C++, consisting of approximately 5,000 lines of code. We use the 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 outlined in Goldberg et al [28: §4].

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. This naïve design of the gossip protocol in our prototype implementation is potentially susceptible to Sybil attacks, since it does not prevent an adversary from joining the gossip network with a large number of identities. We leave the problem of implementing a Sybil-resistant gossip network to future work.

One difference between our implementation and the pseudocode shown in §7 lies in the Binary$BA\star$() function. The pseudocode in Algorithm 8 votes in the next 3 steps after reaching consensus. For efficiency, our implementation instead looks back to the previous 3 steps before possibly returning consensus in a future step. This logic produces equivalent results but is more difficult to express in pseudocode.

Figure 4 shows the parameters in our prototype of Algorand; we experimentally validate the timeout parameters in §10. $h = 80\%$ 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 [41], 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 [18] 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 (§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.

Parameter	Meaning	Value
h	assumed fraction of honest weighted users	80%
R	seed refresh interval (# of rounds)	1,000 (§5.2)
τ_{PROPOSER}	expected # of block proposers	26 (§B.1)
τ_{STEP}	expected # of committee members	2,000 (§B.2)
T_{STEP}	threshold of τ_{STEP} for $BA\star$	68.5% (§B.2)
τ_{FINAL}	expected # of final committee members	10,000 (§C.1)
T_{FINAL}	threshold of τ_{FINAL} for $BA\star$	74% (§C.1)
MaxSteps	maximum number of steps in Binary$BA\star$	150 (§C.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	5 seconds

Figure 4: Implementation parameters.

10 EVALUATION

Our evaluation quantitatively answers the following:

- What is the latency that Algorand can achieve for confirming transactions, and how does it scale as the number of users grows? (§10.1)
- What throughput can Algorand achieve in terms of transactions per second? (§10.2)
- What are Algorand's CPU, bandwidth, and storage costs? (§10.3)
- How does Algorand perform when users misbehave? (§10.4)
- Does Algorand choose reasonable timeout parameters? (§10.5)

To answer these questions, we deploy our prototype of Algorand on Amazon's EC2 using 1,000 m4.2xlarge virtual machines (VMs), each of which has 8 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 [53] 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 equal distribution of money maximizes the number of messages that users need to process. 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.

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

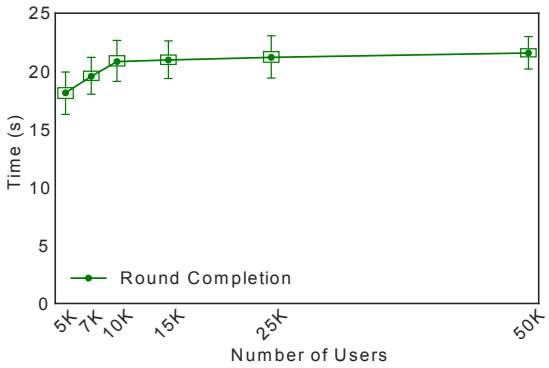

Figure 5: Latency for one round of Algorand, with 5,000 to 50,000 users.

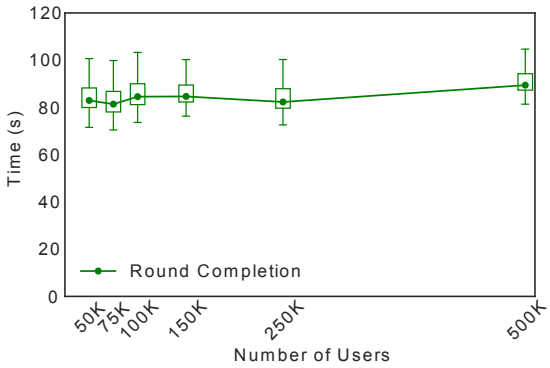

Figure 6: Latency for one round of Algorand in a configuration with 500 users per VM, using 100 to 1,000 VMs.

transactions in well under a minute, and the latency is near-constant as the number of users grows. (Since $\tau_{\text{FINAL}} = 10{,}000$, the time it takes to complete the FINAL step increases until there are 10,000 users in the system; before this point, users are selected more than once and send fewer votes with higher weights.)

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 50,000 to 500,000 (by varying the number of VMs from 100 to 1,000). The latency in this experiment is about 4× higher than in Figure 5, even for the same number of users, owing to the bandwidth bottleneck. However, the scaling performance remains roughly flat all the way to 500,000 users, suggesting that Algorand scales well.

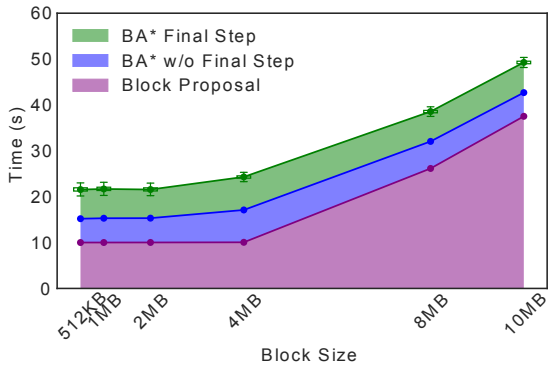

Figure 7: Latency for one round of Algorand as a function of the block size.

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 three parts. Block proposal (§6), at the bottom of the graph, is the time it takes a user to obtain the proposed block. 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. *BA*⋆ except for the FINAL step, in the middle of the graph, is the time it takes for *BA*⋆ to reach the FINAL step. Finally, *BA*⋆'s FINAL step, at the top of the graph, is the time it takes *BA*⋆ to complete the FINAL step. We break out the FINAL step separately because, for the purposes of throughput, it could be pipelined with the next round (although our prototype does not do so).

The results show that Algorand's agreement time (i.e., *BA*⋆) is independent of the block size, and stays about the same (12 seconds) even for large blocks. The throughput can be further increased by pipelining the FINAL step, which takes about 6 seconds, with the next round of Algorand. The fixed time for running *BA*⋆ and the linear growth in block propagation time (with the size of the block) suggest that increasing the block size allows one to amortize the time it takes to run *BA*⋆ to commit more data, and therefore reach a throughput that maximizes the network capability.

At its lowest latency, Algorand commits a 2 MByte block in about 22 seconds, which means it can commit 327 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, Algorand achieves higher throughput at the cost of some increase to latency. For example, with a 10 MByte block size, Algorand commits about 750 MBytes of transactions per hour, which is 125× Bitcoin's throughput.

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 8-core VM was about 40% (most of it for verifying signatures and VRFs),

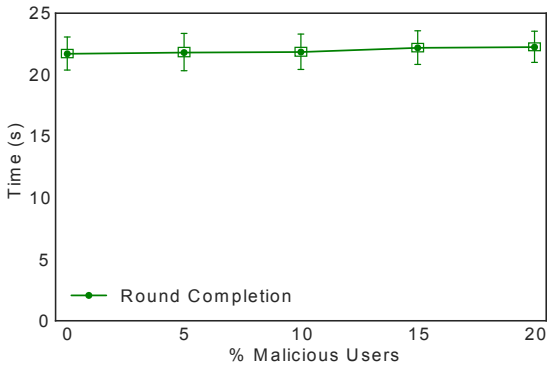

Figure 8: Latency for one round of Algorand with a varying fraction of malicious users, out of a total of 50,000 users.

meaning each Algorand process uses about 6.5% of a core. In terms of bandwidth, each user in our experiment with 1 MByte blocks and 50,000 users uses about 10 Mbit/sec (empirically computed as the total amount of data sent, divided by the duration of the experiment). We note that the communication cost per user is independent of the number of users running Algorand, since users have an expected fixed number of neighbors they gossip messages to, and the number of messages in the consensus protocol depends on the committee size (rather than the total number of users).

In terms of storage cost, Algorand stores block certificates in order to prove to new users that a block was committed. This storage cost is in addition to the blocks themselves. Each block certificate is 300 KBytes, independent of the block size; for 1 MByte blocks, this would be a ~30% storage overhead. Sharding block storage across users (§8.3) reduces storage costs proportionally. For example, sharding modulo 10 would require each user to store, on average, 130 KB for every 1MB block that is appended to the ledger.

10.4 Misbehaving 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 (note that as an optimization, if a user receives to conflicting versions of a block from the highest priority block proposer before the block proposal step is complete, he discards both proposals and starts *BA*⋆ with the empty block). Malicious users that are chosen to be part of the *BA*⋆ committee vote for both blocks. Figure 8 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.

10.5 Timeout parameters

The above results 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} (5 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 [18].

11 FUTURE WORK

This paper focused on the consensus mechanism for committing transactions, and addressing the associated scalability and security challenges. There remain a number of open problems in designing permissionless cryptocurrencies:

Incentives. In order to encourage Algorand users to participate, i.e., be online when selected and pay the network cost of operating Algorand, the system may need to include incentives, possibly in form of a reward mechanism. Designing and analyzing an incentive mechanism includes many challenges, such as ensuring that users do not have perverse incentives (e.g., to withhold votes), and that malicious users cannot "game the system" to obtain more rewards than users who follow the protocol (e.g., by influencing seed selection).

Cost of joining. To join Algorand, new users fetch all existing blocks with their accompanying certificates, which can comprise a large amount of data. Other cryptocurrencies face a similar problem, but since the throughput of Algorand is relatively high, this may create a scalability challenge.

Forward security. Attackers may attempt to corrupt users over time, since identities of committee members are revealed after they send a message. If an attacker manages to obtain enough user keys, he could construct a fake certificate to create a fork. One solution would be for users to forget the signing key before sending out a signed message (and commit to a series of signing keys ahead of time, perhaps using identity-based encryption [11, 20]).

12 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*⋆ 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 125× the throughput of Bitcoin, and scales well to 500,000 users.

ACKNOWLEDGMENTS

Thanks to Iddo Bentov, Ethan Heilman, Jelle van den Hooff, and our shepherd, Robbert van Renesse, for their helpful comments and suggestions. Gilad, Hemo, and Zeldovich were supported by NSF awards CNS-1413920 and CNS-1414119.

REFERENCES

[1] 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.

[2] 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.

[3] I. Bentov, C. Lee, A. Mizrahi, and M. Rosenfeld. Proof of activity: Extending Bitcoin's proof of work via proof of stake. In *Proceedings of the 2014 Joint Workshop on Pricing and Incentives in Networks and Systems*, Austin, TX, June 2014.

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

[5] I. Bentov, P. Hubáček, T. Moran, and A. Nadler. Tortoise and hares consensus: the Meshcash framework for incentive-compatible, scalable cryptocurrencies. Cryptology ePrint Archive, Report 2017/300, Apr. 2017. http://eprint.iacr.org/.

[6] D. J. Bernstein. Curve25519: New Diffie-Hellman speed records. In *Proceedings of the 9th International Conference on Theory and Practice in Public-Key Cryptography (PKC)*, pages 207–228, New York, NY, Apr. 2006.

[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)*, Santa Barbara, CA, Aug. 2001.

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

[13] V. Buterin. Minimal slashing conditions. https://medium.com/@VitalikButerin/minimal-slashing-conditions-20f0b500fc6c, Mar. 2017.

[14] 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)*, pages 524–541, Santa Barbara, CA, Aug. 2001.

[15] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4), Nov. 2002.

[16] J. Chen and S. Micali. Algorand. Technical report, 2017. URL http://arxiv.org/abs/1607.01341.

[17] 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.

[18] 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.

[19] 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.

[20] N. Döttling and S. Garg. Identity-based encryption from the Diffie-Hellman assumption. In *Proceedings of the 37th Annual International Cryptology Conference (CRYPTO)*, pages 537–569, Santa Barbara, CA, Aug. 2017.

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

[22] P. Erdős and A. Rényi. On the evolution of random graphs. *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, 5:17–61, 1960.

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

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

[25] 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.

[26] I. Eyal, A. E. Gencer, E. G. Sirer, and R. van Renesse. Bitcoin-NG: A scalable blockchain protocol. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 45–59, Santa Clara, CA, Mar. 2016.

[27] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. Algorand: Scaling Byzantine agreements for cryptocurrencies. Cryptology ePrint Archive, Report 2017/454, Version 20170924:210956, Sept. 2017. http://eprint.iacr.org/.

[28] 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/.

[29] 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.

[30] S. Higgins. Bitcoin mining pools targeted in wave of DDoS attacks. Mar. 2015. https://www.coindesk.com/bitcoin-mining-pools-ddos-attacks/.

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

[32] 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.

[33] E. Kokoris-Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford. Enhancing Bitcoin security and performance with strong consistency via collective signing. In *Proceedings of the 25th Usenix Security Symposium*, pages 279–296, Austin, TX, Aug. 2016.

[34] 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.

[35] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[36] 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.

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

[38] S. Micali. Fast and furious Byzantine agreement. In *Proceedings of the Innovations in Theoretical Computer Science (ITCS) Conference*, 2017.

[39] 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.

[40] 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.

[41] 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.

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

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

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

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

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

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

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

[49] 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/.

[50] 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.

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

[52] M. Vasek, M. Thornton, and T. Moore. Empirical analysis of denial-of-service attacks in the Bitcoin ecosystem. In *Proceedings of the 18th International Financial Cryptography and Data Security Conference*, Barbados, Mar. 2014.

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

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

ALGORAND AGREEMENT
Super Fast and Partition Resilient Byzantine Agreement

Jing Chen Sergey Gorbunov Silvio Micali Georgios Vlachos
{jing, sergey, silvio, georgios@algorand.com}

April 25, 2018

Abstract

We present a simple Byzantine agreement protocol with leader election, that works under $> 2/3$ honest majority and does not rely on the participants having synchronized clocks. When honest messages are delivered within a bounded worst-case delay, agreement is reached in expected constant number of steps when the elected leader is malicious, and is reached after two steps when the elected leader is honest. Our protocol is resilient to arbitrary network partitions with unknown length, and recovers fast after the partition is resolved and bounded message delay is restored.

We will briefly discuss how the protocol applies to blockchains in a permissionless system. In particular, when an honest leader proposes a block of transactions, the first voting step happens in parallel with the block propagation. Effectively, after the block propagates, a certificate is generated in just one step of voting.

1 Introduction

In this short manuscript, we describe a fast Byzantine agreement protocol with leader election, which is safe even in asynchronous networks. Algorand ledger will be based on the permissionless version of this protocol.[1] In traditional Byzantine agreements, users try to agree on one of their starting values. In a Byzantine agreement with leader election, users try to agree on a value proposed by a leader.

Our protocol is simple and reaches agreement quickly when the network is not partitioned. In particular, it achieves the following desirable properties under $> \frac{2}{3}$ honest majority:

- **Fast Agreement.** When the network has bounded delay —that is, all honest messages propagate in the network within a given time bound—, an agreement is reached in a constant expected time. In particular, when the leader is honest, his proposed value is agreed upon after two steps of communication.
- **Arbitrary Partition Resilience (i.e., Asynchronous Safety).** When the network is partitioned (especially, the Adversary has complete control on message delivery and messages may be delayed arbitrarily long), our protocol ensures safety of the system so that no two honest users will finish the protocol with different outputs.

[1] For historical references, extensions, and evaluation of Algorand ledger we refer the interested readers to [1, 2, 3, 4].

- **Fast Recovery from Network Partition.** After the network recovers from a partition and restores bounded delay, an agreement is reached in constant expected time.

2 Preliminaries

Cryptographic Primitives. We shall rely on two well known cryptographic tools: a hash function H modeled as a random oracle, and digital signatures. More precisely, each player i has a public/secret key pair from a digital signature scheme. Each player holds her secret key privately. Public keys are known to every player in the system. We denote i's signature of a string x by $sig_i(x)$. To ensure signer identities and messages are retrievable from signatures, we define

$$SIG_i(x) \triangleq (i, x, sig_i(x)).$$

The signature scheme is secure under adaptive chosen message attacks and enjoy the following *uniqueness property*: for each public/secret key pair —even the maliciously generated ones— and each message m, there is only one string that is accepted as the signature of m relative to that public key.[2]

Temporary Simplifying Assumptions. In this manuscript, we simplify the protocol description by making the following assumptions, that will soon be relaxed.

1. The setting is permissioned, with a fixed set of users. The set of all players is N, the cardinality of N is $n = 3t + 1$, and the number of malicious players is t.

2. Each user i has a private input v_i at the beginning of the protocol. The set of possible inputs is denoted by V, and there is a special symbol $\perp \notin V$. The users try to agree on a value in V.

3. All players have access to a public random string R, which has been selected randomly and independently of the players' public keys.

The Adversary. The Adversary perfectly coordinates all malicious players. He learns the messages sent by honest players and then chooses the messages sent by the malicious players. However, the Adversary cannot forge honest players' signatures or break the hash function.

From permissioned to permissionless. The protocol described in this manuscript can be generalized to the permissionless setting as in the original Algorand protocol, where the Adversary can corrupt users adaptively and instantaneously, but cannot control more than 1/3 of the total stake in the system. An execution of the permissionless protocol corresponds to one round in the Algorand blockchain, where users agree on a block of transactions. Similarly, the original Algorand paper also describes how the random string R can be generated and updated as the blockchain grows.

[2] Our protocol also works using verifiable random functions (VRFs) [5], which can be constructed under concrete complexity assumptions.

3 The Agreement Protocol

For simplicity, we first describe the protocol in an idealized network setting, Communication Setting 1. Next, we describe the changes that lead to a protocol that achieves the three desirable properties highlighted in our introduction in a concrete network setting, Communication Setting 2.

Communication Setting 1. The players communicate, in steps, over a synchronous propagation network. Honest users send messages at the start of a step and such messages are received by all honest users by the end of the step. Moreover, all messages seen by an honest user i before the start of step s will be seen by all honest users by the end of step s, as user i helps propagate those messages.

Communication Setting 2. The players communicate over a propagation network. Users do not have synchronized clocks, but their individual timers have the same speed. The network may be arbitrarily partitioned for an unknown amount of time, during which the Adversary has full control on the delivery of messages. When the network is not partitioned, a message propagated by an honest user is received by all honest users within time λ. However, the Adversary fully controls the delivery orders of different messages. All messages sent by honest users during a partition are delivered to honest users after the partition is resolved, within time $c\lambda$ for some constant c.

3.1 Notions and Notations

The protocol is a 5-step loop. For conceptual clarity we describe the loop as 5-step periods. Each period has a leader, defined as follows.

Definition 3.1. Credential: *User i's* credential σ_i^p *for a period p is $SIG_i(R, p)$.*

Definition 3.2. Leader: *The* leader ℓ_p *for period p is the user* $\arg\min_{j \in N} H(SIG_j(R, p))$.

When a user i identifies his own leader *for period p, $\ell_{i,p}$, i sets $\ell_{i,p}$ to be the user* $\arg\min_{j \in S_i} H(SIG_j(R, p))$*, where S_i is the set of all users from which i has received valid period-p credentials.*

Our protocol will refer to three types of messages, defined below.

Definition 3.3. Cert-vote: *User i's* cert-vote *for a value v for period p is the signature $SIG_i(v, \text{“}cert\text{”}, p)$.*

We say a user i cert-votes *a value v for period p when he propagates $SIG_i(v, \text{“}cert\text{”}, p)$. We say a user i has* certified *for period p if he has cert-voted a value v for period p.*

Definition 3.4. Soft-vote: *User i's* soft-vote *for a value v for period p is the signature $SIG_i(v, \text{“}soft\text{”}, p)$.*

We say a user i soft-votes *a value v for period p when he propagates $SIG_i(v, \text{“}soft\text{”}, p)$.*

Definition 3.5. Next-vote: *User i's* next-vote *for a value v for period p is the signature $SIG_i(v, \text{“}next\text{”}, p)$.*

We say a user i next-votes *a value v when he propagates $SIG_i(v, \text{“}next\text{”}, p)$.*

3.2 The Protocol in Communication Setting 1

Users start in period 1, and after step 5 of period p moves to step 1 of period $p+1$. User i starts with a private value v_i, and there is a special symbol $\perp$ different from the users' private values.

Period p

STEP 1: [Value Proposal]

- *If* $\Big(p = 1\Big)$ *OR* $\Big((p \geq 2)$ *AND* (i *has received* $2t+1$ *next-votes for* $\perp$ *for period* $p-1)\Big)$, *then* i *proposes* v_i, *which he propagates together with his period* p *credential;*
- *Else if* $\Big(p \geq 2\Big)$ *AND* $\Big(i$ *has received* $2t+1$ *next-votes for some value* $v \neq \perp$ *for period* $p-1\Big)$, *then* i *proposes* v, *which he propagates together with his period* p *credential.*

STEP 2: [The Filtering Step]

- *If* $\Big(p = 1\Big)$ *OR* $\Big((p \geq 2)$ *AND* (i *has received* $2t+1$ *next-votes for* $\perp$ *for period* $p-1)\Big)$, *then* i *identifies his leader* $\ell_{i,p}$ *for period* p *and soft-votes the value* v *proposed by* $\ell_{i,p}$*;*
- *Else if* $\Big(p \geq 2\Big)$ *AND* $\Big(i$ *has received* $2t+1$ *next-votes for some value* $v \neq \perp$ *for period* $p-1\Big)$, *then* i *soft-votes* v.

STEP 3: [The Certifying Step]

- *If* i *sees* $2t+1$ *soft-votes for some value* $v \neq \perp$, *then* i *cert-votes* v.

STEP 4: [The Period's First Finishing Step]

- *If* i *has certified some value* v *for period* p, *he next-votes* v*;*
- *Else he next-votes* $\perp$.

STEP 5: [The Period's Second Finishing Step]

- *If* i *sees* $2t+1$ *soft-votes for some value* $v \neq \perp$ *for period* p *and has not next-voted* v *in Step 4, then* i *next-votes* v.[a]

[a] By the end of Step 5, an honest user i is guaranteed to see $2t+1$ next-votes for some value v, which may or may not be $\perp$. Thus Steps 1 and 2 of the next period is well defined.

In Communication Setting 1, Steps 4 and 5 can be combined into one step. We keep them separate to better align with Communication Setting 2.

The Halting Condition

User i HALTS the moment he sees $2t+1$ cert-votes for some value v for the same period p, and sets v to be his output. Those cert-votes form a *certificate* for v.

3.3 The Protocol in Communication Setting 2

In this setting, each user i keeps a timer $clock_i$ which he resets to 0 every time he starts a new period. As long as i remains in the same period, $clock_i$ keeps counting. The users' individual timers do not need to be synchronized or almost synchronized. We only require they have the same speed.

The halting condition is the same as in Communication Setting 1 and is omitted from the description below.

Period p

All honest users start period 1 at the same time.[a] User i starts period $p \geq 2$ the first moment he receives $2t+1$ next-votes for some value v for period $p-1$, and only if he has not yet started a period $p' > p$.[b] User i sets his *starting value* for period $p \geq 2$, st_i^p, to v. For $p = 1$, $st_i^1 \triangleq \perp$. The moment user i starts period p, he finishes all previous periods and resets $clock_i$ to 0.

STEP 1: [Value Proposal] User i does the following when $clock_i = 0$.

- *If* $\Big(p = 1\Big)$ *OR* $\Big((p \geq 2)$ *AND* (i *has received* $2t+1$ *next-votes for* $\perp$ *for period* $p-1)\Big)$, *then* i *proposes* v_i, *which he propagates together with his period* p *credential;*
- *Else if* $\Big(p \geq 2\Big)$ *AND* $\Big(i$ *has received* $2t+1$ *next-votes for some value* $v \neq \perp$ *for period* $p-1\Big)$, *then* i *proposes* v, *which he propagates together with his period* p *credential.*

STEP 2: [The Filtering Step] User i does the following when $clock_i = 2\lambda$.

- *If* $\Big(p = 1\Big)$ *OR* $\Big((p \geq 2)$ *AND* (i *has received* $2t+1$ *next-votes for* $\perp$ *for period* $p-1)\Big)$, *then* i *identifies his leader* $\ell_{i,p}$ *for period* p *and soft-votes the value* v *proposed by* $\ell_{i,p}$*;*
- *Else if* $\Big(p \geq 2\Big)$ *AND* $\Big(i$ *has received* $2t+1$ *next-votes for some value* $v \neq \perp$ *for period* $p-1\Big)$, *then* i *soft-votes* v.

STEP 3: [The Certifying Step] User i does the following when $clock_i \in (2\lambda, 4\lambda)$.

- *If* i *sees* $2t+1$ *soft-votes for some value* $v \neq \perp$, *then* i *cert-votes* v.

STEP 4: [The Period's First Finishing Step] User i does the following when $clock_i = 4\lambda$.

- *If* i *has certified some value* v *for period* p, *he next-votes* v*;*
- *Else if* $\Big(p \geq 2\Big)$ *AND* $\Big(i$ *has seen* $2t+1$ *next-votes for* $\perp$ *for period* $p-1\Big)$, *he next-votes* $\perp$.
- *Else he next-votes his starting value* st_i^p.

STEP 5: [The Period's Second Finishing Step] User i does the following when $clock_i \in (4\lambda, +\infty)$, until he is able to finish period p.

- *If* i *sees* $2t+1$ *soft-votes for some value* $v \neq \perp$ *for period* p, *then* i *next-votes* v.
- *If* $\Big(p \geq 2\Big)$ *AND* $\Big(i$ *sees* $2t+1$ *next-votes for* $\perp$ *for period* $p-1\Big)$ *AND* $\Big(i$ *has not certified in period* $p\Big)$, *then* i *next-votes* $\perp$.

[a] Even if different users start period 1 hours apart in time, it is as if the network has been partitioned and, once all honest users have started, the protocol guarantees the three desirable properties described in the introduction.

[b] When the network is not partitioned, an honest user always goes through periods in order. During a partition and shortly after a partition is resolved, however, an honest user may see enough next-votes for a value v' for a period $p'-1$ with $p' > p$ and start period p', before he sees enough next-votes for v for period $p-1$. In this case, he will skip period p.

4 Analysis Sketch

Below we sketch the key points for the three desired properties of our protocol in Communication Setting 2, and we will use the following definitions.

Definition 4.1. Potential starting value for period p: *A value v that has been next-voted by $t+1$ honest users for period $p-1$.*

Definition 4.2. Certified value for period p: *A value v that has been cert-voted by $2t+1$ users for period p.*

Definition 4.3. Potentially certified value for period p: *A value v that has been cert-voted by $t+1$ honest users for period p.*

Note that the Adversary can turn a potentially certified value into a certified value, by adding cert-votes of the t malicious users.

Fast Agreement without Partition.

- If the period-1 leader ℓ_1 is honest, then every honest user i identifies ℓ_1 as his leader in Step 2, thus soft-votes the leader's proposed value v. As there are $2t+1$ honest users and they only soft-vote for v, in Step 3 every honest user sees $2t+1$ soft-votes for v by time 3λ, and no other value v' has these many soft-votes. Thus honest users all cert-vote v by time 3λ. Accordingly, all honest users see $2t+1$ cert-votes for v for period 1 and output v by time 4λ.

  Moreover, from the moment the first honest user i finishes the protocol, all honest users finish within time λ, as i has helped propagating the cert-votes he sees.

- If there is no certified value for period 1 (which only happens if the leader ℓ_1 is malicious), all honest users move to period 2 by time 6λ, and they move within time λ apart.

  Indeed, if no honest user has cert-voted in Step 3, then all honest users next-vote $\perp$ (which is their starting values for period 1) in Step 4. Thus they all see these votes and move to period 2 by time 5λ. (They may or may not have finished Step 5.)

  If some honest users have cert-voted in Step 3, then there exists a value v which has $2t+1$ soft-votes and no other value can have these many soft-votes. Thus those honest users have all cert-voted for v, and there are at most $t+1$ of them (otherwise v is potentially certified and the Adversary can make it certified by adding t cert-votes from malicious users). Since those honest users have helped propagating the soft-votes for v by time 4λ, all honest users see $2t+1$ soft-votes for v by time 5λ. Thus they all next-vote v (in Step 4 or 5) by time 5λ, and all see $2t+1$ next-votes for the same value by time 6λ. Note that some honest users may have next-voted for $\perp$ in Step 4 as well, thus there may also exist $2t+1$ next-votes for $\perp$.

- More generally, if there is no certified value for period $p \geq 2$, all honest users move to period $p+1$ by their own time 8λ, and they move within time λ apart.

  The extra 2λ time compared with period 1 is because the honest users start period p not at the same time but within time λ apart. The invariant remains that all honest users finish a step within time λ apart.

  More over, there exist at most two values each of which has $2t+1$ next-votes for period p, and one of them is necessarily $\perp$.

- In each period p, the leader is honest with probability $> 2/3$. If a period $p \geq 2$ is reached and ℓ_p is honest, then all honest users finish the protocol in period p by their own time 6λ, with the same output $v \neq \perp$.

  Indeed, in period $p-1$, if there exists a certified value v, then $v \neq \perp$, at least $t+1$ honest users have cert-voted for v and helped propagating the $2t+1$ soft-votes for v by the end of their Step 3. Thus at least $t+1$ honest users next-voted v in Step 4 and did not next-vote anything else in period $p-1$. The other honest users next-voted v in Step 5 in period $p-1$. So there do not exist $2t+1$ next-votes for $\perp$ and all honest users move to period p with starting value v. In period p, the leader ℓ_p proposes v in Step 1 and all honest users soft-vote v in Step 2. In Step 3, by their own time 4λ, all honest users have cert-voted v. Thus all of them finish the protocol by their own time 6λ, with output v.

  If there is no certified value in period $p-1$, then the leader ℓ_p may propose his private input v_{ℓ_p} or a value $v \neq \perp$ for which she has seen $2t+1$ next-votes from period $p-1$. In the first case, all honest users will follow the leader and soft-vote v_{ℓ_p}; in the second case, all honest users have seen enough next-votes for v and will soft-vote v in Step 2. In both cases, all honest users will cert-vote the same value in Step 3 and finish the protocol with that value.

- Combining the above facts together, if the period-1 leader ℓ_1 is malicious, then the protocol takes in expectation at most 2.5 periods and at most 16λ time. Moreover, all honest users finish within time λ apart.

Arbitrary Partition Resilience. The following properties hold even during a network partition.

- For each period p, at most one value is certified or potentially certified.
- If a value v is potentially certified for period p, then only v can receive $2t+1$ next-votes for period p. Thus, the unique potential starting value for period $p+1$ is v.
- If a period p has a unique potential starting value $v \neq \perp$, then only v can be certified for period p. Moreover, honest users will only next-vote v for period p, so the unique potential starting value for period $p+1$ is v. Inductively, any future periods $p' > p$ can only have v as a potential starting value. Thus, once a value is potentially certified, it becomes the unique value that can be certified or potentially certified for any period, and no two honest users will finish the protocol with different outputs.

Fast Recovery from Network Partition. The following properties hold after a network partition is resolved.

- If some honest user has seen a certificate during the partition, then all honest users will receive the certificate within time λ after the partition is resolved and they will all HALT.
- Else, let p be the highest period that some honest user is working on when the partition is resolved. After time λ, all honest users will also start period p as they receive $2t+1$ next-votes for period $p-1$. Soon after, all honest users will next-vote the same value v (which may be $\perp$) for period p, and they will all start period $p+1$ within time λ apart.
- Once all honest users start the same period p within time λ apart, we are back in the no-partition case.

5 Extensions

Producing a certificate in one voting step. When our protocol is used to implement the Algorand blockchain, the proposed values are hashes of blocks and are propagated in parallel with the actual blocks. Our protocol allows the users to soft-vote for the hashes in Step 2 without seeing the blocks. As hashes and soft-votes are short messages and propagate much faster than blocks, by the time most honest users receive the actual block B, they should have already received $2t+1$ soft-votes for $H(B)$ when the leader is honest. Thus most honest users cert-vote $H(B)$ the moment they receive B, and a certificate is produced in only one voting step after the block is propagated.

Dynamic adversary in permissionless settings. In a permissionless system where the Adversary can corrupt users dynamically, the (small) committees of Steps 4 and 5 may all be corrupted during a partition after sending out their next-votes, and all their messages may be pocketed by the Adversary, in which case their votes may not be propagated to all honest users after the partition is resolved. Since the next-votes are the means of moving to the next period, we introduce new steps and committees in order to make progress after a partition. In particular, in the permissionless protocol, for each period p we add steps $6, 7, 8, \ldots$, where even-numbered steps are essentially copies of Step 4 and odd-numbered steps are essentially copies of Step 5. The corresponding rule for moving to period $p+1$ would be seeing $2t+1$ next-votes for some value v from the same step of period p.

6 Discussions

The Optimality of 2/3 Honest Majority. Following the classic literature on Byzantine agreements, no agreement protocol that works under $\leq 2/3$ honest majority can be partition resilient. Thus our protocol has the optimal dependence on the honesty ratio among all partition-resilient agreement protocols.

Short timers. It is not necessary for a user i's timer to be able to keep track of time forever. Indeed, we only need that users' individual timers can count up to a short fixed interval —the amount of time it takes a user to reach Step 5 after starting a period, when there is no partition. In practice this interval is no more than a few minutes, depending on how fast a block propagates. If a long network partition happens, the timers will all be reset shortly after the partition is resolved and the protocol still achieves the three desired properties given in the Introduction.

Player replaceability. Our protocol is *player-replaceable* as in the original Algorand protocol, which allows us to change the set of users that vote in each step, tolerating an Adversary who is able to corrupt users instantaneously. Indeed, in a permissionless system, we use Algorand's cryptographic self-selection to select small voting committees, and the committee members use ephemeral keys to sign their votes. We will describe the permissionless protocol and the corresponding conditions for committee selection in another manuscript.

References

[1] Jing Chen, Silvio Micali. ALGORAND. In arXiv report http://arxiv.org/abs/1607.01341 Version 9.

[2] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos , Nickolai Zeldovich. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In SOSP '17.

[3] Sergey Gorbunov and Silvio Micali, Democoin: A Publicly Verifiable and Jointly Serviced Cryptocurrency. In Cryptology ePrint Archive, Report 2015/521.

[4] Silvio Micali, ALGORAND: The Efficient and Democratic Ledger. In arXiv report http://arxiv.org/abs/1607.01341 Version 1.

[5] Silvio Micali, Salil Vadhan, and Michael Rabin. Verifiable Random Functions. In FOCS '99.

Vault: Fast Bootstrapping for the Algorand Cryptocurrency

Derek Leung, Adam Suhl, Yossi Gilad, and Nickolai Zeldovich
MIT CSAIL

***Abstract*—Decentralized cryptocurrencies rely on participants to keep track of the state of the system in order to verify new transactions. As the number of users and transactions grows, this requirement becomes a significant burden, requiring users to download, verify, and store a large amount of data to participate.**

Vault is a new cryptocurrency design based on Algorand that minimizes these storage and bootstrapping costs for participants. Vault's design is based on Algorand's proof-of-stake consensus protocol and uses several techniques to achieve its goals. First, Vault decouples the storage of recent transactions from the storage of account balances, which enables Vault to delete old account state. Second, Vault allows sharding state across participants in a way that preserves strong security guarantees. Finally, Vault introduces the notion of stamping certificates, which allow a new client to catch up securely and efficiently in a proof-of-stake system without having to verify every single block.

Experiments with a prototype implementation of Vault's data structures show that Vault's design reduces the bandwidth cost of joining the network as a full client by 99.7% compared to Bitcoin and 90.5% compared to Ethereum when downloading a ledger containing 500 million transactions.

I. Introduction

Cryptocurrencies enable decentralized electronic payments, smart contracts, and other applications. However, supporting a large number of users and transactions will require cryptocurrencies to address two crucial and related bottlenecks: *storage* (how much data every participant needs to store) and *bootstrapping* (how much data every participant has to download to join the system). For example, in Bitcoin [21], a new client that wishes to join the network and verify that it received the correct state must download about 150 GB of data, as of January 2018 [3]. Storage and bootstrapping costs are related because, in a decentralized design, existing nodes must store enough state to help new nodes join the system.

Designing a cryptocurrency whose storage and bootstrapping costs scale well with the number of users and transactions is difficult due to several challenges. First, a cryptocurrency must prevent double-spending—that is, prevent a user from spending the same money twice or issuing the same transaction multiple times. This is typically done by keeping track of past transactions, but doing so is incompatible with good scalability. For instance, Bitcoin stores all past transactions, which does not scale well (costs grow linearly with the number of transactions). As another example, Ethereum [9] does not store all transactions but instead keeps track of the sequence number ("nonce") of the last transaction issued from a given account [28]. This nonce must be stored even if the account has no remaining balance. As a result, this does not scale well either (costs grow linearly with the number of old accounts) and has caused problems for Ethereum when a smart contract inadvertently created many zero-balance accounts [6], [29]. We measure the Ethereum ledger (§VII) and find that 38% of Ethereum accounts have a balance of zero.

Second, a cryptocurrency relies on all participants to check the validity of transactions. This requires the participants to have enough state to validate those transactions. Storing all account balances allows a participant to validate any transaction but requires storage space that grows with the number of accounts. On the other hand, not storing all account balances could imply that fewer participants can vet transactions.

Third, proof-of-stake systems, such as Algorand [14], can provide high transaction throughput. However, such proof-of-stake systems are particularly challenging in terms of bootstrapping cost. Convincing a new participant of the validity of a block in the blockchain requires first convincing them of the balances (stakes) of all users in an earlier block. Convincing a new user of the validity of the latest block thus requires convincing them of the balances of all users at all points in time, starting with the initial genesis block.

Finally, an appealing way to reduce storage and bootstrapping costs is to delegate the job of storing state and certifying future states to a committee whose participants are trusted in aggregate. However, existing systems that take this approach [17], [18], [22] rely on *long-standing* committees known to an adversary. As a result, this adversary may be able to target the committee members, leading to security or availability attacks.

This paper presents Vault, a new cryptocurrency design based on Algorand that addresses the storage and bootstrapping bottlenecks described above. In particular, Vault reduces the bandwidth cost of joining the network as a full client by 99.7% compared to Bitcoin and 90.5% compared to Ethereum when downloading a ledger containing 500 million transactions. Vault builds on Algorand's proof-of-stake consensus protocol and addresses the above challenges of storage and bootstrapping costs using several techniques:

First, Vault *decouples* the tracking of account balances from the tracking of double-spent transactions. Each Vault transaction is valid for a bounded window of time, expressed in terms of the position in the blockchain where the transaction can appear. This allows Vault nodes to keep track of just the transactions that appeared in recent blocks and to forget about all older transactions. The account balance state, on the other hand, is not directly tied to past transactions, and zero-balance accounts can be safely evicted.

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23313
www.ndss-symposium.org

Second, Vault uses an *adaptive sharding scheme* that combines three properties: (1) it allows sharding the account state across nodes so that each node does not need to store the state of all accounts; (2) it allows all transactions to be validated by all nodes, using a Merkle tree to store the balance information; and (3) it adaptively caches upper layers of the Merkle tree so that the bandwidth cost of transferring Merkle proofs grows logarithmically with the number of accounts.

Finally, Vault introduces *stamping certificates* to reduce the cost of convincing new users of a block's validity. The insight lies in trading off the liveness parameter used in selecting a committee to construct the certificate of a new block [14], [22].[1] The stamping certificates are built on top of existing Algorand certificates and have a much lower probability of selecting an online quorum (so in many cases Vault fails to find enough participants to construct a valid certificate) but require fewer participants to form the certificate (thus significantly reducing their size) while still preserving the same safety guarantees (i.e., an adversary still has a negligible probability of corrupting the system). Building an extra layer of stamping certificates allows us to relax liveness for stamping without affecting the liveness of transaction confirmation. Vault's stamping certificates are generated in a way that allows new clients to skip over many blocks in one verification step.

We prototype and benchmark the core of Vault's design, focusing on bootstrapping and storage. Our evaluation shows that Vault's storage and bootstrapping cost is 477 MB for 500 million transactions when account creation and churn rates match those observed in Ethereum in practice. This is a significant reduction compared to existing systems like Ethereum and Bitcoin; with the same 500 million transactions, Ethereum and Bitcoin would require 5 GB and 143 GB respectively. Individual microbenchmarks demonstrate that each of Vault's techniques are important in achieving its performance goals.

The contributions of this paper are:

- The design of Vault, a cryptocurrency that reduces storage and bootstrapping costs by 10.5–301× compared to Bitcoin and Ethereum and that allows sharding without weakening security guarantees.
- Techniques for reducing storage costs in a cryptocurrency, including the decoupling of account balances from double-spending detection and the adaptive sharding scheme.
- The stamping certificate technique for reducing bootstrapping costs in a proof-of-stake cryptocurrency.
- An evaluation of Vault's design that demonstrates its low storage and bootstrapping costs, as well as the importance of individual techniques.

II. Motivation

Vault's goal is to reduce the cost of storage and bootstrapping in a cryptocurrency. There are two significant aspects to this goal, corresponding to two broad classes of prior work.

The first is what we call the "width" of the ledger: how much data does each participant need to store in order to validate transactions (including detecting double-spending)? In Bitcoin, for example, the "width" is the set of all past unspent transactions [21]. Techniques that address the width of a ledger focus on managing the substantial storage costs of keeping the history of all transactions on each client. Vault reduces its "width" by decoupling account state from transaction state (§IV) and by adaptively sharding its state (§V).

The second is what we call the "length" of the ledger: how much data must be transmitted to a new participant as proof of the current state of the ledger? In Bitcoin's case, the proof consists of all block headers starting from the genesis block, chained together by hashes in the block headers, as well as all of the corresponding block contents (to prove which transactions have or have not been spent yet). Techniques addressing the length of the ledger typically allow clients to skip entries when verifying block headers, which reduces the total download cost. Vault reduces its "length" by using stamping certificates to omit intermediate state (§VI).

Table I summarizes Vault's characteristics and compares them with other cryptocurrencies. Bitcoin and Ethereum [9] provide no formal guarantees on the correctness of the latest state. Permissioned cryptocurrencies have low bootstrapping cost but are vulnerable to an adversary which compromises a quorum of permissioned nodes at any point. A system combining OmniLedger [17] and Chainiac [22] lacks single points of failure, but even then an adversary may adaptively compromise a selected committee. Algorand [14] provides strong security guarantees, but its bootstrapping costs grow prohibitively quickly. Vault alone achieves cryptographic security against an adversary that can adaptively compromise users while scaling in both storage and bootstrapping costs. We explore related work in more detail in §VIII.

III. Overview

Vault is a *permissionless*, *proof-of-stake* cryptocurrency that significantly reduces new client bootstrapping costs relative to the state of the art by reducing both steady-state storage costs and the sizes of proofs needed to verify the latest state.

A. Objectives

Suppose Alice is a new participant in Vault who holds the correct genesis block. She wishes to catch up to the latest state and contacts Bob, an existing participant (or perhaps a set of participants). Vault should achieve the following main goals:

- *Efficient Bootstrapping*: If Bob is honest, he should be able to convince Alice that his state is correct and deliver this state using a minimal amount of bandwidth.
- *Complete Bootstrapping*: If Bob is honest, then upon synchronization, Alice has sufficient state to execute the entire protocol correctly. Moreover, Alice should now be able to help other new clients catch up.
- *Safe Bootstrapping*: If Bob is malicious, he should not be able to convince Alice that any forged state is correct.
- *Efficient Storage*: Bob must store a small amount of data to help Alice join the network.

Vault's design also confers additional benefits:

[1] Vault avoids the use of long-standing committees by using Algorand's cryptographic sortition and player-replaceable consensus.

System	Execution State	Proof Size	Bootstrap Security
Bitcoin [21]	UTXOs	Headers + TXs	Probabilistic (heaviest chain wins)
Ethereum [9]	All accounts	Headers + All accounts	Probabilistic (heaviest chain wins)
Permissioned	$\frac{\text{Live accounts}}{\text{Shards}}$	Majority of trust set's signatures	Cryptographic if majority never compromised; none otherwise
OmniLedger [17] + Chainiac [22]	$\frac{\text{UTXOs}}{\text{Shards}}$	$\frac{\text{Headers+Certificates}}{\text{Sparseness}} + \frac{\text{UTXOs}}{\text{Shards}}$	Cryptographic with static attacker; none with adaptive attacker
Algorand [14]	UTXOs	Headers + Certificates + TXs	Cryptographic
Vault	$\frac{\text{Live accounts}}{\text{Shards}}$	$\frac{\text{Headers+Certificates}}{\text{Sparseness}} + \frac{\text{Live accounts}}{\text{Shards}}$	Cryptographic

TABLE I. VAULT COMPARED TO OTHER CRYPTOCURRENCIES. UTXO REFERS TO UNSPENT TRANSACTION OUTPUTS; TX REFERS TO TRANSACTIONS.

- *Charging for Storage*: Adversaries must acquire significant stake to inflate the size of the protocol state.
- *Liveness and Availability*: Despite sharding state across clients, Vault continues to operate even when some users disconnect from the network. Additionally, Vault maintains bootstrap efficiency even when some users lose connectivity after a block is confirmed.

B. Threat Model

Vault should achieve its goals even in the face of adversarial conditions. However, many properties are unachievable given an arbitrarily strong attacker, which can indefinitely drop, delay, and reorder messages on a network [13]. We therefore limit the attacker's power with the following assumptions, inherited from Algorand [14]:

- *Bounded Malicious Stake*: At least some proportion h of all money (the "stake") in Vault is controlled by honest users, where $h > \frac{2}{3}$. Stake sold by a user counts towards h for some duration d (e.g., 48 hours) following the sale.
- *Cryptographic Security*: The adversary has high but bounded computation power. In particular, the adversary cannot break standard cryptographic assumptions.
- *Adaptive Corruptions*: The adversary may corrupt a particular user at any time (given that at no point it controls more than $1 - h$ of the stake in Vault).
- *Weak Synchrony*: The adversary may introduce *network partitions* lasting for a duration of at most b (e.g., 24 hours). During a network partition, the adversary may arbitrarily reschedule or drop any message. The minimum time between network partitions is nontrivial (e.g., 4 hours).

C. Algorand Background

Vault's consensus protocol is based on Algorand, which we briefly review here. All users' clients in Vault agree on an ordered sequence of signed transactions, and this sequence constitutes the cryptocurrency *ledger*. Vault is a permissionless proof-of-stake system, meaning that any user's client, identified by a cryptographic public key, may join the system, and the client of any user who holds money may eventually be selected to append to the ledger. Honest clients listen for new transactions and append recent valid transactions to the ledger.

The frequency at which a user's client is selected is proportional to the user's stake. Sets of transactions are batched into *blocks*. Each block contains a *block header*, which contains a cryptographic commitment to the transaction set. Block headers also contain the cryptographic hash of the previous block in the ledger. Block headers are small, so these hashes allow clients to quickly verify historical transaction data.

Additionally, block headers contain a special pseudorandom *selection seed* Q. Before a client proposes a block, it computes Q in secret, so Q is unpredictable by the rest of the network and partially resistant to adversarial manipulation. As in Algorand, Vault uses Q to seed *Verifiable Random Functions* (VRFs) [20] to implement *cryptographic sortition*. Cryptographic sortition produces a sample of the users in the system, weighted by the stake of their accounts. Each client's membership in the sample remains unknown to an adversary until the client emits a message because a VRF allows the client to compute this membership privately; since VRFs produce a proof of their correctness, any other client can verify this membership. To protect the system against adversaries which corrupt a user after that user is selected, clients sign their messages with ephemeral keys, which they delete before transmission.

Vault uses a Byzantine agreement scheme which operates in *rounds*. Each round, the protocol selects some block proposer which assembles the transaction set and header forming the block, which is broadcast via a peer-to-peer gossip network [12]. Subsequently, the protocol selects a committee which verifies the correctness of the block. To sample users in a manner resistant to adversarial manipulation, committees from round r are seeded with the value of Q from round $r-1$ and weighted by proofs of stake from round $r - b$.

Once clients become confident of a block's confirmation, Vault uses sortition to select a subset of clients to certify the block by signing its receipt (i.e., Algorand's "final" step). The aggregation of these signatures past some secure threshold, along with proofs of stake for each signature, forms a *final certificate* which proves to any client that a block is valid: the Byzantine agreement protocol guarantees that for each round, at most one valid block (or an empty block if the proposer misbehaves) reliably receives this certificate. Given knowledge of only the *genesis* (i.e., the first) block, a new client is convinced that the block from round r is correct if a peer can produce $r - 1$ block headers and the $r - 1$ corresponding certificates of validity.

D. System Design

Figure 1 gives an overview of Vault's data structures, which are the key to Vault's lower storage and bootstrapping costs. The data structures are based around a chain of block headers, shown in the middle of the figure. Each block header consists of four elements: PREVBLOCK (the hash of the previous block), Q (the seed for cryptographic sortition), TXROOT (a Merkle tree commitment [19] to the list of transactions in the

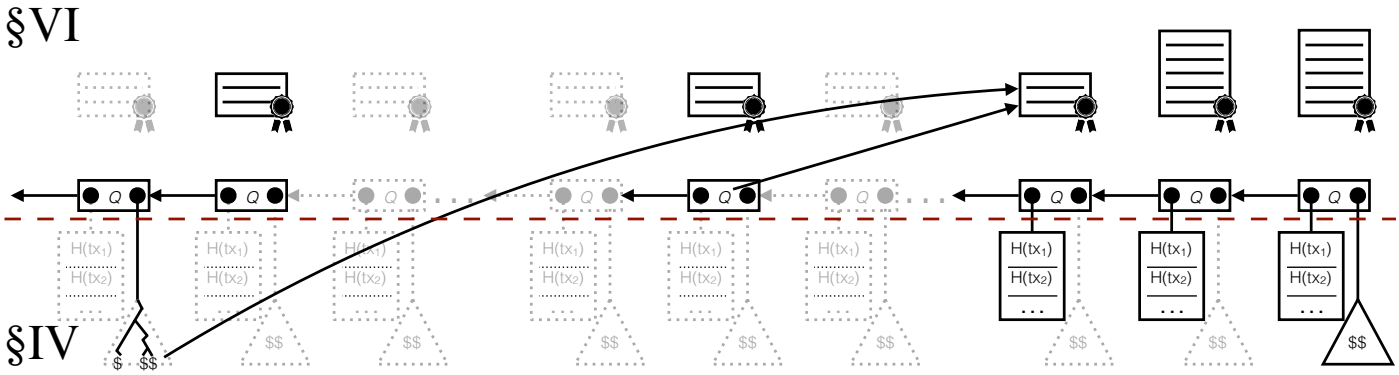

Fig. 1. An overview of the authenticated data structures used in Vault. In this figure, the objects each client stores locally on disk are outlined in solid black, while the objects it may discard are outlined with faint dots. The triangles annotated with "$$" represent the sparse Merkle trees containing account balances, while the bottom row of rectangles annotated with "$H(\mathrm{tx})$" represents the set of transaction hashes in each block. Both the transaction hash set and the balance set are committed to in block headers (the row of rectangles in the middle of the figure); the commitments are represented as solid black dots. In addition, each block header contains Q (i.e., the selection seed), which is computed pseudorandomly and seeded with the previous header's Q-value. The top row of rectangles and seals represent Vault's small stamping certificates and large final certificates; we draw arrows to illustrate how a particular certificate is verified by two block headers. Not shown is Vault's adaptive sharding (§V).

block), and BALROOT (a sparse Merkle tree commitment [8] to the balances of every user after applying transactions).

Every block must follow four rules to be considered valid:

1) Transactions in the block are not expired. Each transaction includes the first and last block number (in the blockchain) where it can appear.
2) After all transactions in the block are executed, no account ends up with a negative balance.
3) Transactions in the block have not been executed before (i.e., have not appeared previously on the ledger).
4) BALROOT correctly reflects all users' balances after applying the block's transactions to the previous block's balances.

In order to check that a new block follows these rules, clients maintain two pieces of state, shown in solid black (as opposed to grayed out) in the bottom half of Figure 1:

- The tree of account balances from the most recent block. This allows a client to ensure that new transactions have sufficient funds (rule 2) and to verify the correctness of the new balance tree (rule 4).
- The lists of transactions from the last few blocks. This allows a client to ensure that a transaction has not appeared previously (rule 3) by checking that a new transaction does not appear in any of the previous transaction lists. To minimize the storage required by these lists, TXROOT commits to a list of transaction hashes, rather than the transactions themselves.

  Clients can discard transaction lists older than a certain threshold, corresponding to the maximum validity interval of a transaction, which we denote $T_{\max}$. Transactions that appeared more than $T_{\max}$ blocks ago will be rejected by rule 1 and need not be tracked explicitly.

§IV describes in more detail how clients check these rules while using a minimal amount of storage. §V further describes Vault's adaptive sharding, which allows clients to store only a subset of the balance tree. These techniques combine to reduce the "width" of Vault's ledger.

Vault uses Algorand's consensus protocol to decide which valid block will be next in Vault's blockchain. The consensus protocol produces a *final certificate* confirming agreement on that block, shown in the top half of Figure 1. These certificates allow a new client to securely join the system and determine which chain of blocks is authentic.

Each certificate consists of a set of signatures (of the block header) by a committee of clients chosen pseudorandomly using cryptographic sortition. In order to verify a certificate, a new client must check that all of the signatures are valid (which is straightforward) and check that the clients whose signatures appear in the certificate were indeed members of the committee chosen by cryptographic sortition (which requires state). Verifying committee membership requires two pieces of state: the sortition seed Q, used to randomize the selection, and the balance tree at BALROOT, used to weigh clients by how much money their users have.

In Algorand's certificate for block r, BALROOT comes from block $r-b$, while Q comes from block $r-1$. This means that, in order to verify block r, the client must first verify block $r-1$ so that the client knows the correct Q for verifying block r's certificate. Furthermore, the committees used for Algorand's certificates are relatively large, so that with high probability there are enough committee members to form a certificate for each block. These certificates are shown with a tall rectangle at the top of Figure 1.

Vault introduces a second kind of certificate, called a *stamping certificate*, which speeds up bootstrapping. The stamping certificate differs from the final certificate in two important ways. First, instead of using Q from the immediately previous block, it uses Q from b blocks ago. (For security, BALROOT must be chosen from b blocks before Q, so this means BALROOT now comes from $2b$ blocks ago.) This allows clients to "vault" forward by b blocks at a time. Second, the stamping certificates use a smaller committee size, which makes the certificate smaller since it contains fewer signatures. The shorter rectangles at the top of Figure 1 represent stamping certificates, with arrows indicating the Q and BALROOT needed to verify them.

Vault sets parameters so that the stamping certificate is just as hard for an adversary to forge as Algorand's original certificates. The trade-off, however, is that in some blocks, there may not be enough committee members to form a valid stamping certificate. To help new clients join the system, every Vault client keeps the stamping certificates for approximately every bth block since the start of the blockchain, along with full Algorand-style certificates for the blocks since the last stamping certificate. Other certificates are discarded (shown as grayed out in Figure 1). §VI-B describes Vault's stamping certificates in more detail, which help Vault shrink the "length" of its ledger.

IV. Efficient Double-Spending Detection

This section describes Vault's design for minimizing the amount of storage required by a client to verify new transactions. To understand the challenges in doing so, consider the key problem faced by a cryptocurrency: *double-spending*. Suppose Alice possesses a single coin which she gives to both Bob and Charlie. A cryptocurrency must reject one of these transactions; if both are accepted, Alice double-spent her coin.

In Bitcoin, each transaction has a set of inputs and outputs. The inputs collect money from previous transactions' outputs, which can then be used by this transaction. The outputs define where the money goes (e.g., some may now be spendable by another user, and the rest remains with the same user). To detect double-spending in this scheme, Bitcoin must determine whether some output has been previously spent or not. Thus, clients must store the set of all unspent transaction outputs.

A more space-efficient approach is to store the balance associated with each user, rather than the set of unspent transactions. For example, Ethereum follows this approach. The cost savings from storing just the balances may be significant: for instance, there are ten times as many transactions in Bitcoin as there are addresses [2], [4].

Switching to a balance-based scheme introduces a subtle problem with transaction replay. If Alice sends money to Bob, Bob may attempt to re-execute the same transaction twice. In Bitcoin's design, this would be rejected because the transaction already spent its inputs. However, in a naïve design that tracked only account balances, this transaction still appears to be valid (as long as Alice still has money in her account), and Bob may be able to re-execute it many times to drain Alice's account.

To distinguish between otherwise identical transactions, Ethereum tags each account with a *nonce*, which acts as a sequence number. When an account issues a transaction, it tags the transaction with its current nonce, and when this transaction is processed, the account increments its nonce. The transactions issued by an account must have sequential nonces. Because of this design, Ethereum cannot delete accounts with zero balance; all clients must track the nonces of old accounts to prevent replay attacks, on the off chance that the account will receive money in the future.

Empty accounts significantly increase the storage cost of Ethereum. Our analysis of its ledger shows that approximately one-third of all Ethereum addresses have zero balance (§VII). Worse, the inability to garbage-collect old accounts constitutes a serious denial-of-service vulnerability: an adversary with a small amount of money may excessively increase the cryptocurrency's storage footprint by creating many accounts. In fact, in 2016 an Ethereum user inadvertently created many empty accounts (due to a bug in Ethereum's smart contract processing) [6], requiring the Ethereum developers to issue a hard fork to clean up the ledger [29].[2]

Alice→Bob:	$30
Issuance:	550
Expiry:	574
Nonce:	8
	Alice

Fig. 2. The format of a Vault transaction from Alice to Bob. In addition to the sender, receiver, and amount, the transaction contains $t_{\text{issuance}}, t_{\text{expiry}}$, and a nonce. A valid transaction contains the sender's digital signature.

At a high level, Vault avoids the problem of storing empty accounts by forcing transactions to expire. The rest of this section describes Vault's solution in more detail.

A. Transaction Expiration

All transactions in Vault contain the fields t_{issuance} and t_{expiry}, which are round numbers delineating the validity of a transaction: blocks older than t_{issuance} or newer than t_{expiry} may not contain the transaction. Moreover, we require that $0 \leq t_{\text{expiry}} - t_{\text{issuance}} \leq T_{\max}$ for some constant $T_{\max}$. This way, a verifying client may detect the replaying of a transaction by checking for its presence in the last $T_{\max}$ blocks. (Transactions still contain a nonce to distinguish between otherwise identical transactions; however, this nonce is ephemeral and not stored.) As a result, clients do not need to track account nonces and can delete empty accounts from the balance tree. Figure 2 shows the format of one transaction.

Requiring transaction lifetimes to be finite means that, if a transaction fails to enter a block before it expires (e.g., because its transaction fee was lower than the current clearing rate), the issuer must reissue the transaction in order for the transaction to be executed. On the other hand, the expiration time ensures that old transactions that failed to enter a block when they were originally issued cannot be re-entered into a block at a much later time by an adversary; i.e., expiration bounds the outstanding liabilities of an account.

Transactions are valid for a window of rounds and not just a single round because clients must account for uncertainty in transaction propagation latencies, transaction fees, and round durations. For example, by the time a transaction propagates through the network, some number of rounds will have passed since the issuer signed it. If the transaction were valid for just one round, the issuer must precisely estimate its propagation latency. Similarly, a temporary spike in the transaction fees could occur due to a burst of high demand. A window of transaction validity amortizes over these uncertainties.

The choice of $T_{\max}$ affects two considerations. The first is that clients must store the last $T_{\max}$ blocks' worth of

[2]Currently, Ethereum transaction fees are high enough to make such attacks unlikely. However, proposed cryptocurrency designs like Algorand [14] aim to support orders of magnitude more throughput, which would lead to lower transaction fees, and which would in turn make such attacks worth considering.

transactions to detect duplicates; a larger $T_{\max}$ increases client storage. (Clients can store transaction hashes instead of the transactions themselves to reduce this cost.) The second is that clients must reissue any transaction that fails to enter a block within $T_{\max}$ (if they still want to issue that transaction). In our experiments, we set $T_{\max}$ to the expected number of blocks in 4 hours (which, based on Algorand's throughput of ∼750 MB/hour [14], suggests at most a few hundred megabytes of recent transaction hashes); we believe this strikes a balance between the two constraints.

Although Vault's design requires that transactions be valid for a short window of time (e.g., 4 hours), Vault is nonetheless compatible with applications that require transactions to be settled far in the future. For instance, payment channels are used to collapse multiple off-chain transactions into a single on-chain *settlement* transaction. By aggregating many off-chain transactions over a long period of time, payment channels can reduce on-chain transaction load. In Vault, a settlement transaction can be postdated into future rounds—e.g., a window of rounds that will appear in a week. Thus, even though the transaction itself is valid for a relatively short window of $T_{\max}$, the payment channel can still aggregate off-chain transactions for an arbitrary time period (e.g., an entire week).

B. Efficient Balance Commitments

To efficiently commit to the large set of balances in BALROOT, Vault clients build Merkle trees [19] over this set. Each leaf in the Merkle tree stores a single account—that is, the public key of the account and the balance for that account. The leaves are sorted by public key. With a Merkle tree, clients may prove that some object exists in a given set using a witness of size $\mathcal{O}(\log n)$, where n is the total number of objects in the set. This allows them to efficiently construct proofs of stake. For example, for a balance set containing 100 million accounts (4 GB of on-disk storage), it suffices for a client to send 1 KB of data to prove its stake against a 32-byte BALROOT in a block header. It is important for the proofs of stake in Vault to be small since a certificate may contain thousands of these proofs (see §VI).

For clients to verify the validity of BALROOT for a new block, BALROOT must be deterministically constructed given a set of balances. As a result, the Merkle leaves are sorted before they are hashed together to create the root. Since a leaf may be deleted when an account balance reaches 0, Vault uses *sparse* Merkle trees [8] to perform balance commitments. A sparse Merkle tree possesses the structure of a prefix trie, which allows us to perform tree insertions and deletions with a Merkle path witness of size $\mathcal{O}(\log n)$. In fact, a witness of size $\mathcal{O}(\log n)$ is sufficient for a client to securely update BALROOT *without* storing the corresponding Merkle tree. We exploit this self-verifying property of Merkle witnesses in §V.

C. Safe and Complete Bootstrapping

With these two mechanisms, it suffices to bootstrap a new verifier by transmitting a commitment to the latest state and the last $T_{\max}$ rounds of transactions.

This scheme is complete because it allows a new verifier to detect double-spending. Suppose the current round is r, and a transaction was confirmed in round r_0. Reading the set of transactions in the last $T_{\max}$ rounds is sufficient to detect that this transaction is a duplicate. Either $r - T_{\max} \leq r \leq r_0$, so it is in this set and therefore invalid, or $r_0 < r - T_{\max}$, which means the transaction must have expired at $r_0 + T_{\max} < r$.

Moreover, this scheme is safe due to the cryptographic integrity property of Merkle trees. For an attacker to forge a cryptocurrency state, it must tamper with the preimage to a Merkle tree, which is computationally reducible to the security of a cryptographic hash function.

V. SHARDING BALANCE STORAGE

As the number of accounts in Vault grows, the cost of storing balances becomes the primary bottleneck. Concretely, each account requires about 40 bytes (32 bytes for a public key and 8 bytes for the balance). This means that if there were 100 million accounts, every Vault client would need to store about 4 GB of data, which may be acceptable (e.g., it is less than Bitcoin's current storage cost). On the other hand, if Vault grew to 10× or 100× more accounts, the storage cost would likely be too high for many clients.

To address this problem, Vault shards the tree of balances across clients, pseudorandomly distributed by the client user's key. Sharding allows clients to deal with large ledger sizes. As an extreme example, consider a system with 100 billion accounts and 1 million online clients. Setting the number of shards to 1,000 would require each client to store approximately 4 GB of data (i.e., $1/1000$th of the balances), rather than the full 4 TB balance tree. Even with a large number of shards in this example, every shard's data is held by about 1000 clients, ensuring a high degree of availability.

One challenge in sharding is that fewer clients now have the necessary data to verify any given new transaction. Existing proposals (like OmniLedger [17]) implement sharding by restricting verifiers' responsibility of preventing double-spending attacks to their own shards. These proposals seem attractive because they reduce not just storage costs but also bandwidth and latency costs, allowing the system to scale throughput arbitrarily. Unfortunately, such schemes are vulnerable to adversaries which control a fraction of the currency. As shard size decreases, so do shards' replication factors, and as a result, transactions in a given shard are verified by a small number of clients. An adversary may own a critical fraction of the stake in a shard by chance, giving it control of the entire shard. Thus, these systems require an undesirable trade-off between scaling and security, which in practice limits the degree of sharding.

Vault's design allows sharding without any reduction in security because all clients retain the ability to verify all transactions. The trade-off comes in terms of increased bandwidth costs during normal operation (which may reduce the system's maximum throughput): as we describe in the rest of this section, with Vault's adaptive sharding, each transaction must include a partial Merkle proof, which grows proportionally to the height of the balance tree.

A. Shard Witnesses

Vault shards the tree of account balances into 2^N pieces by assigning an account to a shard according to the top N bits of the account's public key. A client stores a shard by storing

the balances for the accounts in that shard. Clients store the shard(s) corresponding to their user's public key(s). Clients that join the network, or create a new account, download the corresponding shards from existing peers.

The first challenge is to support sharding without a loss in security by ensuring that all clients can verify all transactions. Recall that the balance set is stored in a sparse Merkle tree (§IV-B) whose root is committed to in the block header. These trees support insertions, updates, and deletions with witnesses of size $\mathcal{O}(\log n)$. To allow a client to check the validity of any transaction in a proposed block (even if that transaction operates on accounts outside of this client's shard), Vault transactions include Merkle witnesses for the source and destination accounts in the previous block's BALROOT. Any client that possesses the previous block's BALROOT can use the Merkle witnesses to confirm the source and destination account balances and thus to verify the transaction.

B. Updating Witnesses

The second challenge is that the structure of the Merkle tree storing account balances might change from the time the transaction is issued to the time the transaction is committed to the ledger, invalidating the transaction's witnesses. Vault clients maintain a pool ("mempool" in Bitcoin terminology) of pending transactions that have not yet been committed. Suppose a client has transaction T_0 in its pool, and in the meantime, a new block is committed to the ledger, containing transaction T_1 but not T_0. Applying T_1 to the Merkle tree causes the tree's internal nodes to be updated (since each node is a hash of its children), and as a result, the witnesses for transaction T_0 are no longer valid. A naïve implementation of Vault would thus be vulnerable to denial-of-service attacks: an adversary can send a tiny amount of money to a victim's account, which invalidates pending transactions issued by the victim, and prevents the victim from spending money.

To address this challenge, Vault leverages the fact that witnesses for a transaction need not be signed along with the rest of the transaction. Since the witnesses are self-certifying, the witnesses can be updated without requiring the issuer to re-sign the entire transaction. Building on this insight, Vault clients update the witnesses of pending transactions when any concurrent transactions are applied to the ledger.

Specifically, suppose that a client has an existing witness for account A (which might be the source or destination account associated with T_0), and makes a concurrent change to account B (which might be associated with T_1). Each witness is a path from a leaf node in the Merkle tree (e.g., accounts A or B) to the root of the Merkle tree. For any two witnesses, there must be an intersection point (the lowest node in the tree where the two witnesses overlap). To update the existing witness for account A after a change to account B, Vault re-computes B's witness with the new leaf value for B, then finds the intersection between A's and B's witnesses, and finally splices the top part of B's new witness into A's witness. This produces a fresh witness for A that is valid after B is modified.

The witness update algorithm described above can be executed by any client, even if the client does not store the shards associated with any of the transactions. It relies only on the fact that the newly committed transactions have fresh witnesses, which is ensured by the client that proposed this block of transactions in the first place.

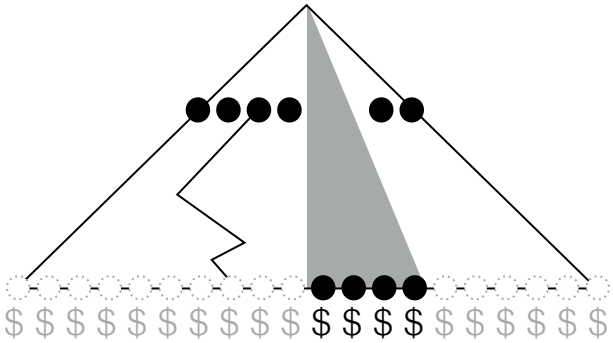

Fig. 3. An illustration of a single Vault shard and the balance Merkle tree. Dots here represent Merkle nodes, and the "$" symbols represent account balances. The solid black dots and dark "$" symbols represent the balances which are part of the shard (the shaded gray triangle), while those in gray represent the parts of the tree which are not. The row of black dots in the middle represent the frontier of Merkle nodes stored by all clients regardless of shard assignment. The jagged line connecting one of these nodes to an unstored leaf represents the Merkle witness necessary for updating a balance.

C. Adaptive Sharding: Truncating Witnesses

The third challenge is that witnesses increase transaction size. For example, if the transaction size is 250 bytes (on par with Bitcoin), and there are 100 billion accounts in the system, a single Merkle witness will hold 37 sibling nodes in expectation, which is 1.2 KB. Two witnesses would introduce 2.4 KB of overhead per transaction—almost an $11\times$ increase. Note that the inclusion of Merkle witnesses increases *bandwidth* but not *storage* costs: since all blocks are certified by an honest committee, verifiers discard the witnesses after they recompute BALROOT. Thus, the trade-off applies only to the bandwidth costs of broadcasting transactions during rounds.

To manage the overhead of larger witnesses, clients store (in addition to their shards) an intermediate frontier that cuts across the Merkle tree—roughly speaking, the subset of tree nodes at some depth. Storing this frontier allows clients to verify *partial* witnesses, which prove the path from a leaf node to the frontier, rather than all the way to BALROOT. Figure 3 illustrates one such partial witness.

We can quantify the trade-off between transaction size and the cost of storing the frontier. Moving the frontier up in the tree by one level (i.e., going from the nodes in the frontier to their parents) increases the length of a partial Merkle witness by a single sibling. Moving the frontier up in the tree by one level also halves its size. Vault can thus tune the trade-off between the size of partial Merkle witnesses in each transaction and the amount of storage required for the frontier.

If the frontier lies in the dense region of the Merkle prefix tree (towards the top of the tree), the shape of the frontier is simple: it involves all the Merkle nodes at a given *level*. However, if the frontier lies near the leaves of the Merkle prefix tree (near the bottom), a client cannot simply store all the nodes at a given level, as the layers are larger than the balance set (owing to the sparseness of the Merkle tree). Instead, these frontiers assume a "jagged" shape; they are defined as the nodes which sit at a fixed *height* from the bottom of the tree.

To update a node in the frontier, it suffices for a client to observe a witness and follow these two rules: (1) if the witness increases the height of a frontier node, the client replaces that frontier node with its children (which were present in the witness); and (2) if the witness decreases the height of a frontier node, the client replaces that frontier node with its parent (if it did not previously store the parent). Note that the length of the witness alone is sufficient for determining whether an insertion, an update, or a deletion occurred.

By application of the coupon collector's problem[3], we see that if there are approximately $n \log n$ account balances, then in expectation the last dense layer is of depth n [5]. For example, if there are 100 billion $\approx 2^{37}$ accounts, then layer $n = 32$ is the last dense one.

D. Safe and Complete Bootstrapping

Vault's adaptive sharding construction maintains the cryptographic integrity of Merkle trees, as any verifier can construct the Merkle proof of any account's stake on demand by concatenating the prefix of the proof, which the verifier stores, with the suffix of the proof, which is provided by the spender and dynamically updated by the verifier. Therefore, this construction is safe and complete.

VI. Succinct Ledger Certificates

Bootstrapping a new client in a proof-of-stake system like Algorand requires transferring a significant amount of data to the new client, due to two factors. First, the confirmation of each block depends on the state of the system at the time the block was proposed. For instance, as mentioned in §III-C, the Algorand committee that forms the final certificate of a block is selected based on the random seed Q from the previous block. Thus, to verify the correctness of block r, a new client must first verify the correctness of block $r-1$ in order to obtain the correct Q value for verifying block r. Second, in Algorand's design, the certificate confirming a block consists of a large number of signatures, reflecting the large committee size. This arrangement is shown at the top of Figure 4.

Vault addresses this problem using two techniques. First, Vault introduces a *stamping* certificate that can be verified using state from b and $2b$ blocks ago rather than the state from 1 block ago. This allows clients to "leapfrog" by b blocks at a time instead of having to verify every single block in the blockchain. Vault uses Algorand's cryptographic sortition to privately select a committee for stamping certificates in a way that does not reveal committee membership to an adversary in advance. This ensures that an adversary cannot selectively corrupt members of this committee to falsify a certificate. Certificate signatures use the ephemeral keys of each committee member; a committee member destroys its keys before it broadcast its signature. This is shown in the middle of Figure 4 and described in more detail in §VI-A.

Second, Vault uses a smaller committee size to generate the stamping certificates, which reduces the size of the certificates themselves (since they contain fewer signatures).[4] To ensure that a smaller committee does not give the adversary a higher probability of corrupting the committee, Vault requires a much larger fraction of expected committee members to vote in order for the stamping certificate to be valid. This means that, with significant probability, the committee fails to gather enough votes to form a stamping certificate. However, this is acceptable because new clients have two fallback options: they can either verify Algorand's full certificate, or they can verify a stamping certificate for a later block and backtrack using PrevBlock hashes in the block headers. This arrangement is shown at the bottom of Figure 4 and described in §VI-B.

A. Leapfrog Protocol

To enable leapfrogging, Vault constructs a sortition committee for the stamping certificate of block r using the seed Q from block $r-c$, where $c \geq 1$ is some constant. For security, the proof-of-stake balances must be selected from b blocks before the seed Q, so they are chosen from block $r-c-b$. Members of this committee wait for consensus on block r, and once consensus is reached, they broadcast signatures for that block (after deleting the corresponding ephemeral signing key), along with proofs of their committee selection. The set of these signatures forms the stamping certificate for BlockHeader_r.

As mentioned above, this committee is, in principle, known as soon as block $r-c$ has been agreed upon. However, the committee is selected in private using cryptographic sortition, and honest clients do not reveal their committee membership until they vote for block r, which prevents an adversary from adaptively compromising these committee members.

Now each certificate depends on two previous block headers: Certificate_r depends on Q from BlockHeader_{r-c} and BalRoot from $\text{BlockHeader}_{r-c-b}$. Moreover, Certificate_r validates BlockHeader_r, which itself contains the value of Q used for Certificate_{r+c} and the value of BalRoot used for $\text{Certificate}_{r+c+b}$.

To optimize for a new client catching up on a long sequence of blocks starting with the genesis block, we set $c = b$, so that the client does not need to validate separate blocks for Qs and BalRoots. This reduces the bootstrapping bandwidth by a factor of b, since a new client needs to download and authenticate every bth block header and certificate.

To ensure that any client can help a new peer bootstrap, all clients store the block header and certificate for blocks at positions that are a multiple of b. Additionally, to ensure that the base case is true, the first $2b$ blocks in Vault are predetermined to be empty. Finally, to quickly catch up after momentarily disconnecting from the network, clients keep the previous $2b$ block headers at all times.

Choosing b. Vault's choice of b trades off the weak synchrony assumption (i.e., partitions last for periods shorter than b) against d, the speed at which sold stake becomes malicious. We briefly justify our choice of b below; we refer the reader to Algorand's security analysis [15] for a formal treatment.

[3] The coupon collector's problem can be stated as follows: suppose there are n distinct types of coupons. A drawn coupon's type is chosen uniformly at random. How many draws are required to collect one coupon of each type?

[4] Note that multi-signatures [1] would not significantly reduce the size of the certificates since the certificate needs to include a proof of cryptographic sortition (VRF) and a partial Merkle proof for each committee member whose signature appears in the certificate, which cannot be aggregated away.

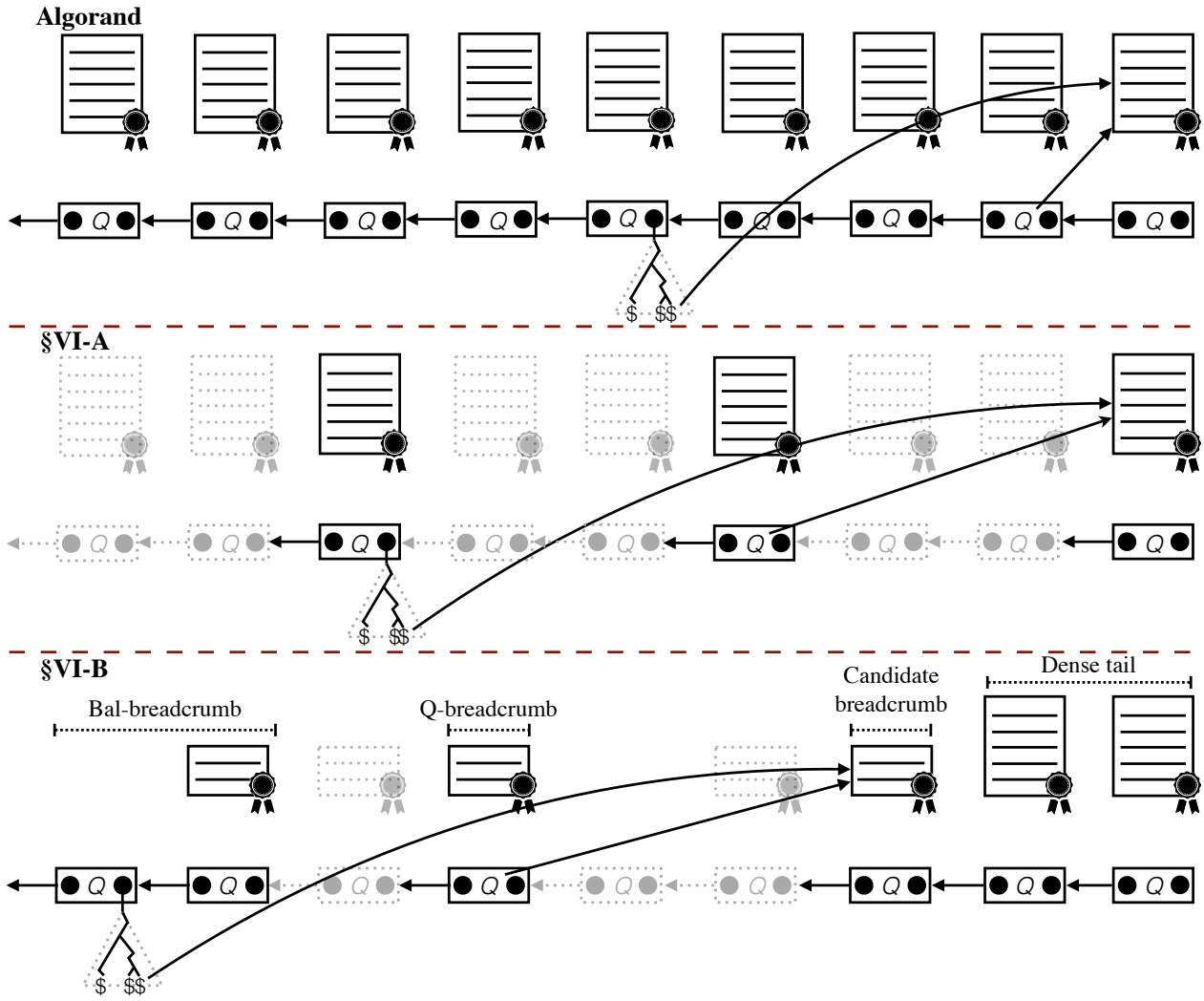

Fig. 4. Two optimizations used to reduce the bandwidth needed to prove validity of the latest state. In this figure, $b = c = 3$; as before, objects that clients can discard are outlined with dots. The top figure depicts the basic ledger data structure without any optimization: a large final certificate authenticates each block header, and each certificate depends on the Q value immediately before it and the proofs of stake b blocks ago. The next figure shows the additional stamping certificate chain with the leapfrog optimization: each leapfrogging certificate depends on the value of Q from b blocks ago and the balances from $2b$ blocks ago (§VI-A). The bottom figure shows stamping committee optimization used to reduce the size of certificates (§VI-B). It illustrates the candidate, Q-, and Bal-breadcrumbs which consist of a small stamping certificate and block header along with a tail of block headers. It shows the dense tail formed by the large final certificates that prove validity of the current block. (The figure omits the frozen breadcrumbs, which are farther back in the chain.)

On the one hand, suppose the adversary partitions the network for more than b blocks starting at round r'. Then the adversary may predict the value of Q at round $r' + b$ and then manipulate its public keys at round r' such that it controls both the proposer and the committee at round $r' + b + 1$. In this way, the adversary confirms two different blocks and thus creates a fork at round $r' + b + 1$. Therefore, b must be large enough to resist complete partitions.

On the other hand, suppose a rich, honest user sells off 50% of the stake in Vault at round r'. A few rounds after the user completes the sale, a poor adversary corrupts this user, who by chance controls a supermajority of the committee at round $r' + b + 1$. Then again the adversary gains control of the ledger. Although this adversary controls little of the system's *current* stake, it controls much of the system's *past* stake. As a result, b must be small enough to allow honest users to finish participating in Vault after selling off their stake.

Since c introduces an extra delay to certificate creation, for security we require that not $b \leq d$ but instead $b + c \leq d$, and since we set $c = b$ we require that $2b \leq d$. At Vault's highest level of throughput, $2b = d = 2880$ corresponds to about two days' worth of blocks.

B. Stamping Committees

Algorand's consensus protocol requires thousands or tens of thousands of signatures to produce a final certificate for a block. This threshold is very high because Algorand guarantees a very low rate of failure in terms of liveness and safety. A failure in liveness prevents a block from being confirmed, while a failure in safety may produce a ledger fork.

As with final certificates, a stamping committee threshold should be set sufficiently high such that an adversary cannot gather the signatures required to trick a new client into accepting a forged ledger fork with high probability. Since

adversaries know when they are selected for a leadership in advance, and a certificate must be secure for all time, we must keep a strict safety threshold.

Although we cannot relax safety, we can greatly relax liveness. Suppose a new client has already verified the block headers for blocks r and $r+b$, using stamping certificates, but there was no stamping certificate produced for block $r+2b$ due to relaxed liveness requirements. If there was a stamping certificate produced for block $r+2b-1$, the new client can efficiently verify that stamping certificate and block instead.

Specifically, the new client can ask an existing peer for the headers of blocks $r-1$ and $r+b-1$ and efficiently verify them by checking PREVBLOCK hashes in blocks r and $r+b$ respectively. Since headers are relatively small, this costs the client little bandwidth. We use the term *breadcrumb* to denote this chain of PREVBLOCK pointers from a stamping certificate to an earlier block header. Figure 4's bottom row shows two breadcrumbs: one that required backtracking by one block (for BALROOT), and one that required no backtracking (for Q).

If the stamping certificate at $r+2b-1$ also failed to form, Vault repeats this process to find the latest block before $r+2b$ that did have a stamping certificate. If no such block exists in a b-block interval, Vault falls back to a full Algorand certificate.

Given that stamping certificate creation may occasionally fail, each breadcrumb must contain a small "tail" of block headers which are required to certify the two subsequent breadcrumbs produced at most b and $2b$ blocks ahead, respectively. Since block headers are relatively small (less than 256 bytes), the cost of storage here is low (less than 1.3 MB for $b = 1440$). As clients observe the confirmation of new blocks and the successful creation of new stamping certificates, they update their state so as to minimize the sizes of these tails. Clients must also hold a *dense tail* of block headers and final certificates at the end of the ledger for each block after the last header for which a stamping certificate was produced. Vault clients discard this dense tail whenever new stamping certificates are successfully created.

C. Safe and Complete Bootstrapping

Ledger certificates completely commit to the state of the ledger by signing a Merkle commitment to this state (§IV-B). Therefore, it suffices to show that this state is the correct one.

While Q is usually unpredictable and random, an adversary may introduce bias into its value during network partitions. Given this bias, Vault requires a safety failure rate of 2^{-100} for both its final and stamping certificates. However, with a relaxed liveness assumption, we can decrease stamping certificate size by at least an order of magnitude.

For example, with an honesty rate of $h = 80\%$, a final certificate requires a threshold of 7,400 signatures. If we allow stamping certificates to fail to form 65% of the time, then it suffices to have a threshold of 100 signatures (out of a suitably smaller committee). Applying the stamping optimizations allows clients in Vault to verify the latest block header in a 10-year old ledger by downloading 365 MB or less. Appendix §A analyzes stamping certificate size given other settings of the honesty and liveness failure rates.

D. Storage Requirements

We next describe the objects that clients keep to allow convincing a new client that the ledger state is valid, and we describe the invariants that apply to each of these components. An algorithm for keeping these objects up-to-date follows.

Proof-objects stored. The bottom of Figure 4 illustrates the objects that a Vault client needs to store.

- The *dense tail* is the set of all headers and full final certificates since the candidate breadcrumb.
- The *candidate breadcrumb* is the breadcrumb with the last-observed stamping certificate. The candidate breadcrumb is tentative and may be overwritten by a "better" breadcrumb (i.e., a more recent breadcrumb which makes the candidate breadcrumb obsolete). This breadcrumb is never more than b blocks ahead of the Q-breadcrumb.
- The *Q-breadcrumb* is the breadcrumb with the stamping certificate immediately preceding the candidate breadcrumb. This breadcrumb's certificate has been fixed as no subsequent certificate may be better than this. However, its tail of block headers may not yet be trimmed.
- The *Bal-breadcrumb* is the breadcrumb with the stamping certificate immediately preceding the Q-breadcrumb. Like the Q-breadcrumb, its certificate is final and unchanging. Moreover, its tail remains "minimal" as new certificates are seen. In other words, it maintains the shortest tail such that the following conditions are true:
  1) It contains the block header needed to authenticate the Q-breadcrumb's certificate's Q-value.
  2) It contains the block header needed to authenticate the candidate breadcrumb's certificate's proofs of stake.
- *Frozen breadcrumbs* are all of the earlier breadcrumbs, from the start of the blockchain to the Bal-breadcrumb. As the blockchain grows (adds new blocks with their respective certificates), breadcrumbs "graduate" from being a candidate breadcrumb, to a Q-breadcrumb, to a Bal-breadcrumb, and eventually to the set of frozen breadcrumbs. It is these frozen breadcrumbs that allow a new client to quickly catch up from the initial genesis block to recent blocks.

Keeping proof-objects up-to-date. All clients maintain a proof of the ledger's latest block. A client mutates its proof on observing two events from the cryptocurrency network:

- When a client observes a block and a full final certificate, it appends them to its dense tail.
- When a client observes a new stamping certificate later than its candidate breadcrumb, it deletes the final certificate of the "stamped" round and all older final certificates. Moreover, it moves ownership of the corresponding block headers to the new stamping certificate, which becomes the "new" breadcrumb. Next,
  - If the new breadcrumb's round is not more than b blocks greater than the Q-breadcrumb's, the client replaces its candidate breadcrumb with the new breadcrumb, transferring ownership of the block headers.
  - Otherwise, the client:

1) Freezes the Bal-breadcrumb, adding it to the set of frozen breadcrumbs.
2) Sets the Q-breadcrumb as the new Bal-breadcrumb. This breadcrumb's tail becomes trimmable.
3) Sets the candidate breadcrumb as the new Q-breadcrumb. This breadcrumb's position is now optimal and fixed (assuming that stamping certificates are received in order).
4) Sets the new breadcrumb as the candidate breadcrumb.

Finally, it "trims" the tail of the Bal-breadcrumb to keep its length minimal.

VII. Evaluation

The primary question that our evaluation answers is, "How effective is Vault at reducing the bandwidth cost of helping a new client join the network?" §VII-B presents the results.

To understand why Vault achieves a reduction in bandwidth, we further answer three questions targeted at each of Vault's techniques, as follows. Recall that two components contribute to bootstrapping costs: the state needed to execute the protocol and the bandwidth required to convince a new client that this state is correct.

- *Balance Pruning*: How much does transaction expiration reduce storage cost by? (§VII-C)
- *Stamping Certificates*: What are the cost savings of using Vault's sparse sequence of stamping certificates for bootstrapping? (§VII-D)
- *Balance Sharding*: What are the trade-offs involved in sharding Vault's balance sets? (§VII-E)

A. Experimental Setup

To answer the questions above, we implemented the data structures needed to execute the Bitcoin, Ethereum, Algorand, and Vault protocols. However, we have not integrated these data structures into their respective systems; in particular, we have not evaluated real-time characteristics of Vault, such as its transaction latency or throughput. We vary transaction volumes between 50 and 500 million transactions, and we fill all blocks with the maximum number of transactions given a fixed block size. (As of February 2018, there are around 300 million transactions in Bitcoin [4] and around 150 million transactions in Ethereum [10].) We ignore the storage cost of auxiliary data structures required to efficiently update a protocol's state; for example, we do not implement database indexes.

Algorand uses a transaction format similar to Bitcoin's. We consider only simple transactions with the form of one input and two outputs (one to the receiver and the other to self).

The ratio of unique accounts to transactions on Ethereum is around 15% [11], [10] as of January 1, 2018. Additionally, we obtained the Ethereum ledger up to this date by synchronizing a Parity [23], [24] Ethereum client (in `fatdb` mode). Our analysis of the Ethereum state indicates that around 38% of all accounts have no funds and no storage/contract data (i.e., only an address and a nonce). For Ethereum and Vault, we fix the account creation rate at 15% and the churn rate at 38%.

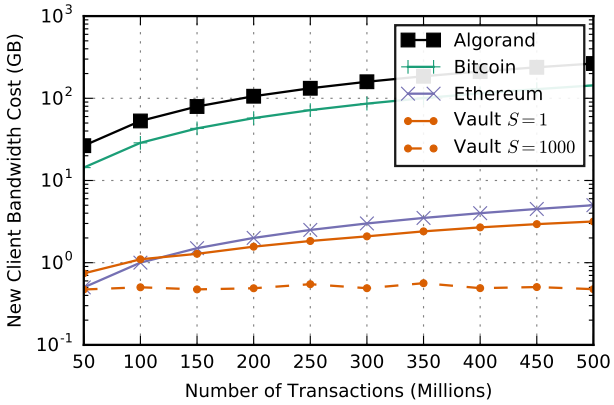

Fig. 5. An end-to-end comparison of cryptocurrency bootstrapping costs. Compared to Bitcoin and Algorand, Vault and Ethereum reduce storage costs by one to two orders of magnitude. Vault outperforms Ethereum at 150 million transactions because it can delete old accounts. Sharding Vault with a factor of 1,000 reduces the costs of storing balances to a negligible amount, and the total storage cost remains low (below 500 MB) even with 500 million transactions on the ledger. Note that the y-axis is logarithmic.

Other than to count the number of empty accounts, we do not consider the costs in Ethereum which result from per-account data storage or from smart contracts.

We instantiate the following parameters both in Algorand and in Vault:

- 80% of the stake in the system is honest ($h = 0.8$).
- Stake sold off by a later-corrupted user counts towards h for $d = 48$ hours.
- Network partitions last for at most 2 days. (Recall that during a network partition, an adversary may arbitrarily reschedule and drop any message.) This implies that the leapfrogging interval is $b = 1440$ rounds (§VI-A).
- The maximum transaction lifetime is $T_{max} = 4$ hours. This keeps the cost of storing the hashes of recent transactions to the hundreds of megabytes.
- Stamping certificates fail to form at a rate of 65%. This implies that a certificate contains $T_{\text{stamping}} = 100$ signatures, and a stamping sortition produces $\tau_{\text{stamping}} = 120$ committee members in expectation (§A).
- The size of a block is 10 MB. (Lower block sizes are possible; these reduce both throughput and latency.)

We use S in the rest of this section to denote the number of shards in Vault.

B. End-to-End Evaluation

Figure 5 presents the results of an end-to-end evaluation of Bitcoin, Ethereum, Algorand, and Vault (with sharding factors of $S = 1$ and $S = 1000$).

Algorand's storage cost exceeds that of Bitcoin. Every transaction that Bitcoin stores must also be stored by Algorand. In addition, supporting secure bootstrapping in Algorand incurs an additional cost ranging from 4 to 47 GB, growing linearly with the number of confirmed transactions in the system.

Figure 5 shows clear gains in storing the set of account balances rather than the set of transactions. Vault and Ethereum,

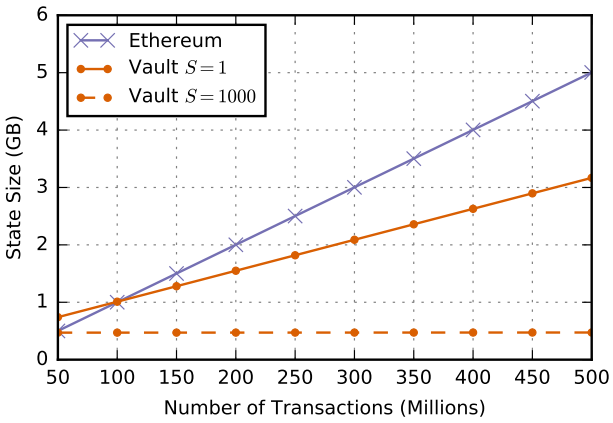

Fig. 6. A comparison of steady-state storage costs between Vault and Ethereum, given an account churn rate of 38%. Pruning empty balances allows Vault to store less data than Ethereum past around 100 million transactions, even as Vault must store its recent transaction log. Note that the line associated with $S = 1000$ grows linearly but appears flat due to its low slope.

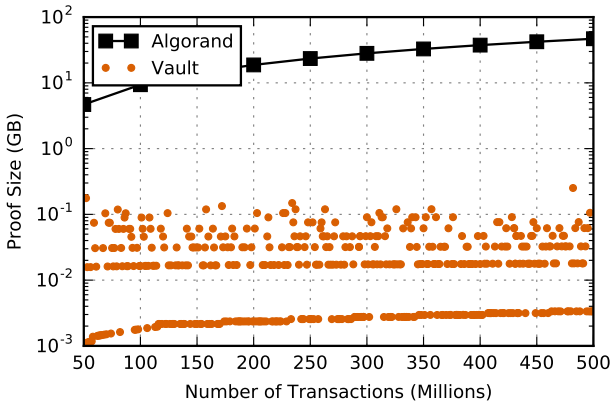

Fig. 7. A comparison between the certificate chain sizes in both Vault and Algorand. The proof size is 2—4 orders of magnitude smaller in Vault, and its size does not exceed 1 GB up to 500 million transactions. Usually, the size of this proof remains under 100 MB. In the plot, proof sizes cluster around several bands, which correspond to the number of final certificates present in the dense tail. The lowest band grows linearly with the number of stamping certificates that were formed. Note that the y-axis is logarithmic.

which both store account balances, outperform Algorand and Bitcoin by 1 to 2 orders of magnitude. This holds both because the set of balances is much smaller than the set of all transactions, and also because an individual balance entry is smaller than a transaction itself. Given that we only consider simple transactions with one input and two outputs, we expect more complex transactions to amplify this effect.

Moreover, at 150 million transactions, Vault outperforms Ethereum even without sharding. This follows from the fact that Vault may delete accounts with no balance, which reduces overall storage cost by about 38%. However, before 150 million transactions, the fixed cost of storing the recent transactions outweighs these savings. We note that throttling the throughput of Vault or reducing T_{max} decreases this cost.

Finally, sharding Vault reduces storage even more significantly. However, sharding is no "free lunch"; it increases the sizes of transactions and thus the steady-state bandwidth cost of propagating them to the entire network (§VII-E).

C. Balance Pruning

To evaluate the efficiency of Vault's balance pruning technique, we compare the storage footprint of Vault's balance set (again sharded at factors of 1 and 1000) against Ethereum's. Since Vault also requires a log of the recent transaction history to detect double-spending, we include these costs as well.

Figure 6 shows that the ability to prune the balance tree significantly reduces the ledger's storage costs at scale. Initially, Vault clients must hold the past 9.6 million transaction hashes to enforce transaction expiration, which costs around 307 MB of overhead (if transaction expiration T_{max} is set to correspond to 4 hours). However, past 150 million transactions, holding the set of account balances dominates the cost of detecting double-spending. Since Ethereum clients cannot garbage collect the 38% of empty accounts in their balance trees, they must store these accounts in perpetuity.[5] Maintaining a log of recent transactions constitutes a constant storage cost, while the overhead of storing empty accounts grows linearly as the system continually processes new transactions.

[5] We speculate that the use of Ethereum's smart contracts to programmatically create temporary accounts only exacerbates this problem. Efficient garbage collection implies cheap temporary account creation.

D. Stamping Certificates

Next, we evaluate how efficiently a client can prove the validity of its state to a new peer. We measure the amount of data transferred for the certificate chain in Vault and compare it against the data transferred for the certificate chain in Algorand. Since the creation of stamping certificates in Vault is non-deterministic, we evaluate the amount of data transferred using fine-grained steps for the independent variable (number of transactions processed) to capture these effects.

Figure 7 reveals that the overhead of the certificate and header storage cost becomes significant in Algorand. To catch up to a ledger with 500 million transactions, a client must download around 47 GB of data.

In contrast, Vault's proofs are much smaller even though the use of a balance-based ledger increases the size of certificates (by including partial Merkle proofs); these proofs are almost always less than 100 MB in size. Two factors decrease the size of these proofs. First, the chain of certificates is much sparser. On average, downloading an extra stamping certificate allows a client to validate an additional $b = 1440$ blocks. Second, each individual stamping certificate is small. Instead of 7,400 signatures, each certificate consists of 100 signatures.

This evaluation demonstrates that, without stamping certificates, certificates would dominate the data required for bootstrapping. A 3.4 GB state size for balances matters little if 47 GB is necessary to prove its validity. Reducing the proof overhead to less than 100 MB allows Vault to securely bootstrap new clients with modest bandwidth cost.

E. Balance Sharding

Under sharding, we would like to determine how decreasing the overhead of storing the intermediate frontier in the balance tree (§V-C) increases the size of transactions. We fix

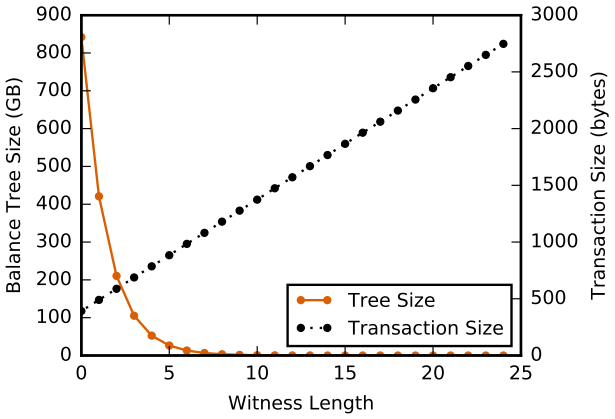

Fig. 8. Cost of storing sharded account balances and transaction sizes given some setting of the witness length. The tree stores 10 billion accounts, divided into $S = 1000$ shards. Increasing transaction size linearly (i.e. extending the Merkle witness) decreases storage overhead exponentially.

the number of accounts to be 10 billion and the number of shards to be $S = 1000$. Figure 8 illustrates this interaction.

We see that shard size is not the limiting factor. With $S = 1000$, each client stores only 10 million accounts per shard, which costs less than 500 MB each. Instead, sharding costs are dwarfed by the overhead of keeping the internal Merkle nodes on the frontier, which allow clients to verify transactions in other shards without needing to receive the entire Merkle path as part of the transaction. On the one hand, all clients may simply store all leaf Merkle nodes, which adds nothing to transaction overhead but also reduces storage cost by only a small amount: storing the set of balances along with the Merkle frontier costs each client almost 1 TB. On the other hand, the exponential fanout of the sparse Merkle tree provides diminishing returns on storing each subsequent layer; extending the Merkle witness by one hash halves the storage footprint of the Merkle nodes. Eventually, storage costs converge to the size of a shard.

VIII. Related Work

In this section we contrast Vault with related systems and discuss their compatibility with Vault's techniques.

A. Steady-State Savings: The "Width" Approach

Several cryptocurrencies adopt mechanisms which reduce the amount of bandwidth needed to join the protocol (as a verifier) and the amount of storage needed to run the protocol.

Ethereum [9] summarizes account balances and other state into a short digest using Patricia Merkle Trees [16]. Each block commits to the root of the current tree, which allows new clients to obtain balance state from any untrusted node and then verify this state against a known Merkle root. To prevent an attacker from replaying a transaction issued by a user, users embed a sequence number (a *nonce*) in each transaction. Ethereum clients must track the last nonce issued by each account in the balance tree, even if the account is empty; otherwise, an old transaction could be replayed (e.g., if an empty account receives a deposit in the future). As we note in §IV, this means that Ethereum's storage cost grows with the number of all accounts that ever existed, leaving Ethereum vulnerable to denial-of-service attacks that create many temporary accounts. By decoupling account balances from tracking double-spent transactions (§IV), Vault prevents storage costs from growing with the number of old accounts. We believe that Vault's decoupling can be adopted by Ethereum to avoid unbounded storage for old accounts: by mandating expiration dates in all transactions, Ethereum can allow nodes to delete old state.

OmniLedger [17] *shards* its ledger by users' public keys, running Byzantine agreement on many ledgers in parallel. OmniLedger performs load balancing across each shard to improve throughput and reduce bandwidth and storage costs proportionally to the number of shards. Sharding allows OmniLedger to scale horizontally. However, OmniLedger requires a long-standing committee to run the PBFT [7] protocol to establish consensus on the ledger's state; this leaves it vulnerable to a strong adversary which may quickly corrupt validators. Moreover, its shard size and thus scalability is sensitive to the proportion of malicious users. Unfortunately, adapting OmniLedger to use Algorand in place of PBFT is non-trivial: it is unclear how Algorand's distinctly sampled committees, which persist not for an epoch but rather change with each message, can exchange state across shards without excessive bandwidth cost. Vault's adaptive sharding (§V) reduces the storage cost per participant and remains secure against an adversary that can quickly corrupt users, but its throughput does not increase with sharding.

An alternative approach to reducing the ledger's "width" is to record fewer transactions on the ledger. The Lightning Network [26] establishes payment channels between pairs of users. Participants in the channel post collateral on the ledger and then exchange transactions off the ledger to record their debts. As a result, by posting only two transactions on the ledger, a pair of participants may process arbitrarily many off-ledger transactions in a payment channel as long as it contains sufficient collateral to absorb them. One advantage of this scheme is that participants do not need to broadcast transactions that take place within a payment channel. We discuss how payment channels can be implemented on top of Vault in §IV-A. However, payment channels come with limitations: they support only pairwise payment relationships, the collateral posted by each participant limits channel capacity, and their incentives assume that participants always act to maximize their payout. A key benefit of Vault is that its storage costs scale with the total number of accounts, rather than the number of transactions, which allows for more transactions to take place "on chain" without the limitations of payment channels.

MimbleWimble [25] uses an accumulator-like *signature sinking* scheme to "compact" blocks together according to the amount of work proved in the block header. Combined blocks eliminate transaction outputs which have been spent, reducing the state a verifier is required to download. Switching to a balance-based scheme like Vault's may allow MimbleWimble to increase its compaction savings by committing not to the set of unspent transactions but rather the current set of balances.

Observe that Vault can safely prune old balances by forcing transactions to expire quickly (§IV), and Vault can shard balances over the nodes in the network for a decrease in storage cost (§V). These schemes work independently of Algorand's

specific characteristics (including proof of stake). Any cryptocurrency which maps cryptographic identifiers to balances and commits to them in a Merkle tree can transparently leverage these techniques.

B. Short Proofs of State: The "Length" Approach

Other cryptocurrencies observe that a small block header is often sufficient evidence of a block's validity. Therefore, they reduce the cost of verifying the block header sequence by shortening it. This allows clients to efficiently prove the validity of their state at any particular point in time.

Like Vault's stamping certificates, Chainiac's [22] Collective Signing (CoSi) [27] scheme allows a committee of verifiers to jointly produce a proof that a particular block is correct. As in Vault, verifying committees for some block also certify the correctness of blocks into the future; upon observing a block confirmation, committees produce forward links to the block. Since these links are arranged in a skiplist-like configuration, they allow verifiers to quickly bootstrap to the current state. However, Chainiac's scheme is inherently vulnerable to an adversary that can adaptively corrupt users because its committees are not secret. Sometime after the protocol designates a committee, an adversary which compromises this committee can forge a proof that a false view of the ledger is valid and thus deceive new clients into accepting a bogus state. Since the committees that produce Vault's certificate signatures are secretly selected and emit exactly one message, Vault's certificates resist attacks from adversaries that can adaptively corrupt clients. Chainiac may thus replace its committees with Vault's committees to improve its bootstrapping security.

MimbleWimble also reduces the length of the ledger. Blocks with more work supersede prior blocks with less work; since an adversary must possess significant processing power to attack these blocks, the proof of work requirements increase the new verifiers' *confidence* in these blocks. As in Bitcoin, this approach does not produce a *proof* of blocks' correctness, since an adversary that controls the network can prevent a user from ever observing the block with the largest amount of work. Vault builds on Algorand for reaching consensus, which ensures safety (no forks) with cryptographic guarantees even in the presence of network partitions.

We observe that in a permissioned cryptocurrency, where a supermajority of ledger writers are trusted, a signed checkpoint suffices to convince a new verifier that the state is correct [7]. Stellar [18] can be thought of in similar terms, where a core node will accept a checkpoint from nodes in its quorum set. Vault targets a permissionless setting where users do not configure trusted sets of known writers or trusted core nodes. As a result, Vault authenticates checkpoint signatures using cryptographic sortition, based on techniques from Algorand.

In general, permissionless cryptocurrencies where randomly sampled committees execute a consensus algorithm (e.g., Chainiac) can use Vault's stamping certificates transparently (§VI). Simply embed the stamping certificates' seed Q (or an existing randomness source) into blocks, and use a Sybil-resistant mechanism (e.g., weigh members via proof of work/stake) to select stamping committees.

IX. Conclusion

Vault is a new cryptocurrency design based on Algorand that reduces storage and bootstrapping costs. Vault achieves its goals using three techniques: (1) transaction expiration, which helps Vault decouple storage of account balances from recent transactions and thus delete old account state; (2) adaptive sharding, which allows Vault to securely distribute the storage of account balances across participants; and (3) stamping certificates, which allow new clients to avoid verifying every block header, and which reduce the size of the certificate. Experiments demonstrate that Vault reduces the storage and bootstrapping cost for 500 million transactions to 477 MB, compared to 5 GB for Ethereum and 143 GB for Bitcoin.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Aniket Kate, for helping us improve this paper. We also thank Georgios Vlachos for assisting us with our security analysis of the certificates. Additionally, we thank David Lazar, Anish Athalye, and the rest of the MIT PDOS group for providing feedback on drafts of this paper. This work was supported by NSF awards CNS-1413920 and CNS-1414119.

References

[1] M. Bellare and G. Neven, "Multi-signatures in the plain public-key model and a general forking lemma," in *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Oct.–Nov. 2006, pp. 390–399.

[2] BitInfoCharts, "Bitcoin rich list," https://bitinfocharts.com/top-100-richest-bitcoin-addresses.html, 2018.

[3] Blockchain Luxembourg S.A., "Blockchain size," https://blockchain.info/charts/blocks-size, 2018.

[4] ——, "Total number of transactions," https://blockchain.info/charts/n-transactions-total, 2018.

[5] G. Blom, L. Holst, and D. Sandell, *Problems and Snapshots from the World of Probability*. Springer Science & Business Media, 2012.

[6] V. Buterin, "Security alert [11/24/2016]: Consensus bug in geth v1.4.19 and v1.5.2," https://blog.ethereum.org/2016/11/25/security-alert-11242016-consensus-bug-geth-v1-4-19-v1-5-2/, 2016.

[7] M. Castro and B. Liskov, "Practical Byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems*, vol. 20, no. 4, Nov. 2002.

[8] R. Dahlberg, T. Pulls, and R. Peeters, "Efficient sparse Merkle trees: Caching strategies and secure (non-)membership proofs," in *Proceedings of the 21st Nordic Conference on Secure IT Systems*, Nov. 2016, pp. 199–215.

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

[10] Etherscan, "Ethereum transaction chart," https://etherscan.io/chart/tx, 2018.

[11] ——, "Ethereum unique address growth rate," https://etherscan.io/chart/address, 2018.

[12] K. R. Fall and W. R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 2011.

[13] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," in *Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS)*, Atlanta, GA, Mar. 1983, pp. 1–7.

[14] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling Byzantine agreements for cryptocurrencies," in *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, Oct. 2017, pp. 51–68.

[15] ——, "Algorand: Scaling Byzantine agreements for cryptocurrencies," Cryptology ePrint Archive, Report 2017/454, May 2017, http://eprint.iacr.org/.

[16] D. E. Knuth, *The art of computer programming*. Pearson Education, 1997, vol. 3.

[17] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, and B. Ford, "OmniLedger: A secure, scale-out, decentralized ledger," Cryptology ePrint Archive, Report 2017/406, Feb. 2018, http://eprint.iacr.org/.

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

[19] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Proceedings of the 7th Annual International Cryptology Conference (CRYPTO)*, Santa Barbara, CA, Aug. 1987, pp. 369–378.

[20] 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.

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

[22] K. Nikitin, E. Kokoris-Kogias, P. Jovanovic, N. Gailly, L. Gasser, I. Khoffi, J. Cappos, and B. Ford, "Chainiac: Proactive software-update transparency via collectively signed skipchains and verified builds," in *Proceedings of the 26th USENIX Security Symposium*, Vancouver, Canada, Aug. 2017, pp. 1271–1287.

[23] Parity Technologies, "Parity," https://www.parity.io/, 2017.

[24] ——, "Github - paritytech/parity: Fast, light, robust Ethereum implementation," https://github.com/paritytech/parity, 2018.

[25] A. Poelstra, "Mimblewimble," Oct. 2016, https://download.wpsoftware.net/bitcoin/wizardry/mimblewimble.pdf.

[26] J. Poon and T. Dryja, "The Bitcoin Lightning network: Scalable off-chain instant payments," Jan. 2016, https://lightning.network/lightning-network-paper.pdf.

[27] E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, P. Jovanovic, L. Gasser, N. Gailly, I. Khoffi, and B. Ford, "Keeping authorities "honest or bust" with decentralized witness cosigning," in *Proceedings of the 37th IEEE Symposium on Security and Privacy*, San Jose, CA, May 2016, pp. 526–545.

[28] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, pp. 1–32, 2014.

[29] ——, "State trie clearing (invariant-preserving alternative)," https://github.com/ethereum/EIPs/blob/master/EIPS/eip-161.md, 2016.

APPENDIX

A. Stamping Certificate Security Analysis

Recall that the security of a certificate is equivalent to the security of the committee that produced it. We first define two desirable properties of Vault's stamping certificates:

Definition. A certificate has a *safety failure rate* of f_s if over all committees produced by cryptographic sortition, the probability that an adversary can obtain two distinct and validating certificates for a given block is f_s.

Definition. A certificate has a *liveness failure rate* of f_l if over all committees produced by cryptographic sortition, the probability that the honest users fail to produce a certificate for a given block is f_l.

In these committees, an honest verifier releases its signature after it sees a block confirmation. Confirmed blocks are fork-safe, so if one honest verifier sees a block, all other honest verifiers will only see that block. Thus, the following holds:

Observation. *In Vault, a stamping certificate with one honest signature is valid.*

Now let T_{stamping} be the threshold of signatures needed to produce a valid stamping certificate, and let τ_{stamping} be the number of committee members elected in expectation to produce this certificate. Moreover, let γ and β be the actual number of honest and malicious users elected to some committee. We obtain two desirable properties, as follows:

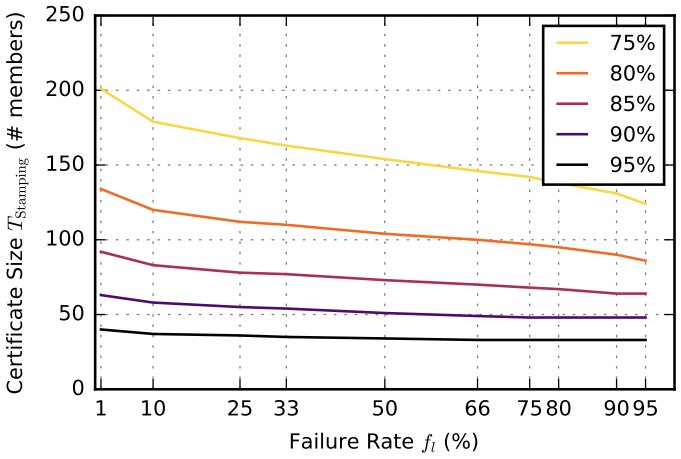

Fig. 9. Certificate sizes required to guarantee various liveness failure rates for a fixed safety failure rate of 2^{-100}, given assumptions on the amount of honest stake in the system. Honesty assumptions have a significant effect on certificate size, which in turn influences the liveness failure rate.

Corollary. *For Vault to produce certificates with a safety failure rate of f_s and liveness failure rate of f_l, we must set τ_{stamping} as follows:*

$$\Pr[\gamma < T_{\text{stamping}}] \leq f_l \tag{1}$$
$$\Pr[\beta \geq T_{\text{stamping}}] \leq f_s \tag{2}$$

For simplicity, suppose that there are an arbitrarily large number of currency units in the system, and let each user in the system own one unit of currency. If h is the proportion of honest users in the system and τ is the expected number of selected users following a cryptographic sortition, we have that the chance of sampling exactly k honest users is

$$\Pr[\gamma = k] = \frac{(h\tau)^k}{k!} e^{h\tau}$$

while the chance of sampling exactly k malicious users is

$$\Pr[\beta = k] = \frac{((1-h)\tau)^k}{k!} e^{(1-h)\tau}.$$

which follows from the application of the binomial theorem.

From Equation 1 and Equation 2 it follows that the following conditions must both hold:

$$\sum_{k=0}^{T_{\text{stamping}}-1} \frac{(h\tau_{\text{stamping}})^k}{k!} e^{h\tau_{\text{stamping}}} \leq f_l \tag{3}$$
$$\sum_{k=T_{\text{stamping}}}^{\infty} \frac{((1-h)\tau_{\text{stamping}})^k}{k!} e^{(1-h)\tau_{\text{stamping}}} \leq f_s \tag{4}$$

Then it is evident that $\tau_{\text{stamping}} = 120, T_{\text{stamping}} = 100$ satisfy these conditions with $h = 0.8, f_s = 2^{-100}, f_l = 0.65$.

Figure 9 illustrates the effects of changing f_l for various values of h, fixing $f_s = 2^{-100}$.

Pixel: Multi-signatures for Consensus

Manu Drijvers
DFINITY

Sergey Gorbunov
Algorand and University of Waterloo

Gregory Neven
DFINITY

Hoeteck Wee*
Algorand and CNRS, ENS, PSL

Abstract

In Proof-of-Stake (PoS) and permissioned blockchains, a committee of verifiers agrees and sign every new block of transactions. These blocks are validated, propagated, and stored by all users in the network. However, posterior corruptions pose a common threat to these designs, because the adversary can corrupt committee verifiers after they certified a block and use their signing keys to certify a different block. Designing efficient and secure digital signatures for use in PoS blockchains can substantially reduce bandwidth, storage and computing requirements from nodes, thereby enabling more efficient applications.

We present Pixel, a pairing-based forward-secure multi-signature scheme optimized for use in blockchains, that achieves substantial savings in bandwidth, storage requirements, and verification effort. Pixel signatures consist of two group elements, regardless of the number of signers, can be verified using three pairings and one exponentiation, and support non-interactive aggregation of individual signatures into a multi-signature. Pixel signatures are also forward-secure and let signers evolve their keys over time, such that new keys cannot be used to sign on old blocks, protecting against posterior corruptions attacks on blockchains. We show how to integrate Pixel into any PoS blockchain. Next, we evaluate Pixel in a real-world PoS blockchain implementation, showing that it yields notable savings in storage, bandwidth, and block verification time. In particular, Pixel signatures reduce the size of blocks with 1500 transactions by 35% and reduce block verification time by 38%.

1 Introduction

Blockchain technologies are quickly gaining popularity for payments, financial applications, and other distributed applications. A blockchain is an append-only public ledger that is maintained and verified by distributed nodes. At the core of the blockchain is a consensus mechanism that allows nodes to agree on changes to the ledger, while ensuring that changes once confirmed cannot be altered.

In the first generation of blockchain implementations, such as Bitcoin, Ethereum, Litecoin, the nodes with the largest computational resources choose the next block. These implementations suffer from many known inefficiencies, low throughput, and high transaction latency [18, 28, 52]. To overcome these problems, the current generation of blockchain implementations such as Algorand, Cardano, Ethereum Casper, and Dfinity turn to proofs of stake (PoS), where nodes with larger stakes in the system —as measured, for instance, by the amount of money in their account— are more likely to participate in choosing the next block [22, 25, 31, 34, 36, 41, 50]. Permissioned blockchains such as Ripple [57] and Hyperledger Fabric [4] take yet another approach, sacrificing openness for efficiency by limiting participation in the network to a selected set of nodes.

All PoS-based blockchains, as well as permissioned ones, have a common structure where the nodes run a consensus sub-protocol to agree on the next block to be added to the ledger. Such a consensus protocol usually requires nodes to inspect block proposals and express their agreement by digitally signing acceptable proposals. When a node sees sufficiently many signatures from other nodes on a particular block, it appends the block to its view of the ledger.

Because the consensus protocol often involves thousands of nodes working together to reach consensus, efficiency of the signature scheme is of paramount importance. Moreover, to enable outsiders to efficiently verify the validity of the chain, signatures should be compact to transmit and fast to verify. Multi-signatures [37] have been found particularly useful for this task, as they enable many signers to create a compact and efficiently verifiable signature on a common message [16, 42, 61, 62].

The Problem of Posterior Corruptions. Chain integrity in a PoS blockchain relies on the assumption that the adversary controls less than a certain threshold (e.g., a third) of the total stake; an adversary controlling more than that fraction

*Authors are listed alphabetically.

may be able to fork the chain, i.e., present two different but equally valid versions of the ledger. Because the distribution of stake changes over time, however, the real assumption behind chain integrity is not just that the adversary *currently* controls less than a threshold of the stake, but that he never did so *at any time in the past*.

This assumption becomes particularly problematic if stake control is demonstrated through possession of signature keys, as is the case in many PoS and permissioned blockchains. Indeed, one could expect current stakeholders to properly protect their stake-holding keys, but they may not continue to do so forever, especially after selling their stake. Nevertheless, without additional precautions, an adversary who obtains keys that represent a substantial fraction of stake at *some* point in the past can compromise the ledger at *any* point in the future. The problem is further aggravated in efficient blockchains that delegate signing rights to a small committee of stakeholders, because the adversary can gain control of the chain after corrupting a majority of the committee members.

Referred to by different authors as *long-range attacks* [21], *costless simulation* [55], and *posterior corruptions* [12], this problem is best addressed through the use of *forward-secure* signatures [3,9,43,48]. Here, each signature is associated with the current time period, and a user's secret key can be updated in such a way that it can only be used to sign messages for future time periods, not previous ones. An adversary that corrupts an honest node can therefore not use the compromised key material to create forks in the past of the chain.

1.1 Our Results

We present the Pixel signature scheme, which is a pairing-based forward-secure multi-signature scheme for use in PoS-based blockchains that achieves substantial savings in bandwidth and storage requirements. To support a total of T time periods and a committee of size N, the multi-signature comprises just two group elements and verification requires only three pairings, one exponentiation, and $N-1$ multiplications. Pixel signatures are almost as efficient as BLS multi-signatures, as depicted in Figure 1, but also satisfy forward-security; moreover, like in BLS multi-signatures, anybody can non-interactively aggregate individual signatures into a multi-signature.

Our construction builds on prior forward-secure signatures based on hierarchical identity-based encryption (HIBE) [15, 19,23,27] and adds the ability to securely aggregate signatures on the same message as well as to generate public parameters without trusted set-up.

We achieve security in the random oracle model under a variant of the bilinear Diffie-Hellman inversion assumption [11, 15]. At a very high level, the use of HIBE techniques allows us to compress $O(\log T)$ group elements in a tree-based forward-secure signature into two group elements, and secure aggregation allows us to compress N signatures under N public keys into a single multi-signature of the same size as a single signature.

To validate Pixel's design, we compared the performance of a Rust implementation [2] of Pixel with previous forward-secure tree-based solutions. We show how to integrate Pixel into any PoS blockchain. Next, we evaluate Pixel on the Algorand blockchain, showing that it yields notable savings in storage, bandwidth, and block verification time. Our experimental results show that Pixel is efficient as a stand-alone primitive and in use in blockchains. For instance, compared to a set of $N = 1500$ tree-based forward-secure signatures (for $T = 2^{32}$) at 128-bit security level, a single Pixel signature that can authenticate the entire set is 2667x smaller and can be verified 40x faster (c.f. Tables 1 and 3). Pixel signatures reduce the size of Algorand blocks with 1500 transactions by $\approx 35\%$ and reduce block verification time by $\approx 38\%$ (c.f. Figures 4 and 5).

1.2 Related Work

Multi-signatures can be used to generate a single short signature validates that a message m was signed by N different parties [6, 10, 14, 33, 37, 45, 46, 51, 53], Multi-signatures based on the BLS signature scheme [14, 16, 17, 56] are particularly well-suited to the distributed setting of PoS blockchains as no communication is required between the signers; anybody can aggregate individual signatures into a multi-signature. However, these signatures are not forward-secure.

Tree-based forward-secure signatures [9, 38, 43, 48] can be used to meet the security requirements, but they are not very efficient in an N-signer setting because all existing constructions have signature size at least $O(N \log T)$ group elements, where T is an upper bound on the number of time periods. Some schemes derived from hierarchical identity-based encryption (HIBE) [15, 19, 23] can bring that down to $O(N)$ group elements, which is still linear in the number of signers.

The only forward-secure multi-signature schemes that appeared in the literature so far have public key length linear in the number of time periods T [47] or require interaction between the signers to produce a multi-signature [58], neither of which is desirable in a blockchain scenario. The forward-secure multi-signature scheme of Yu et al. [64] has signature length linear in the number of signers, so is not really a multi-signature scheme.

Combining the generic tree-based forward-secure signature scheme of Bellare-Miner [9] with BLS multi-signatures [14, 17] gives some savings, but still requires $O(T)$ "certificates" to be included in each multi-signature. Batch verification [8] can be used to speed up verification of the certificates to some extent, but does not give us any space savings. Compared with existing tree-based forward-secure signatures in [9,38,43,48], our savings are two-fold:

- we reduce the size of the signature set for N commit-

scheme	key update	sign	verify	$\|\sigma\|$	$\|pk\|$	$\|sk\|$	forward security
BLS multi-signatures [14, 16, 56]	–	1 exp	2 pair	1	1	$O(1)$	no
Pixel multi-signatures (this work)	2 exp	4 exp	3 pair + 1 exp	2	1	$O((\log T)^2)$	yes

Figure 1: Comparing our scheme with BLS signatures. Here, "exp" and "pair" refer to number of exponentiations and pairings respectively. T denotes the maximum number of time periods. We omit additive overheads of $O(\log T)$ multiplications. The column "key update" refers to amortized cost of updating the key for time t to $t+1$. The columns $|\sigma|$, $|pk|$, and $|sk|$ denote the sizes of signatures, public keys, and secret keys, respectively, in terms of group elements. Aggregate verification for N signatures requires an additional $N-1$ multiplications over basic verification.

tee members from $O(N \log T)$ group elements[1] to $O(1)$ group elements; and

- we reduce the verification time from $O(N)$ exponentiations to $O(1)$ exponentiation and $O(N)$ multiplications.

1.3 Paper Organization

The rest of this paper is organized as follows:

- In Section 2, we give a high level technical description of our new pairing-based forward-secure multi-signature scheme.
- In Sections 4 and 5, we describe the scheme in details. We prove the security of the construction in the random oracle model under a variant of a bilinear Diffie-Hellman inversion problem.
- In Section 6, we explain how to apply Pixel to PoS blockchains to solve posterior corruptions.
- In Section 7, we evaluate the efficiency savings for storage, bandwidth, and block verification time from using Pixel on the Algorand PoS blockchain.
- In Section 8, we describe various extensions to the scheme.
- In Appendix C, we perform theoretical efficiency analysis of Pixel.

2 Technical Overview

Our construction builds on prior forward-secure signatures based on hierarchical identity-based encryption (HIBE) [15, 19,23,27] and adds the ability to securely aggregate signatures on the same message as well as to generate public parameters without trusted set-up.

[1] Each tree-based signature comprise $O(\log T)$ group elements corresponding to a path in a tree of depth $\log T$ (see Section 7 for details), and there are N such signatures, one for which committee member.

Overview of our scheme. Starting with a bilinear group $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t)$ with $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_t$ of prime order q and generators g_1, g_2 for $\mathbb{G}_1, \mathbb{G}_2$ respectively, a signature on $M \in \mathbb{Z}_q$ at time t under public key g_2^x is of the form:

$$\sigma = (\sigma', \sigma'') = (h^x \cdot F(t,M)^r, g_2^r) \in \mathbb{G}_1 \times \mathbb{G}_2$$

where the function $F(t,M)$ can be computed with some public parameters (two group elements in $\mathbb{G}_1$ in addition to $h \in \mathbb{G}_1$) and r is fresh randomness used for signing. Verification relies on the relation:

$$e(\sigma', g_2) = e(h,y) \cdot e(F(t,M), \sigma'')$$

and completeness follows directly:

$$\begin{aligned} e(\sigma', g_2) &= e(h^x \cdot F(t,M)^r, g_2) \\ &= e(h^x, g_2) \cdot e(F(t,M)^r, g_2) \\ &= e(h, g_2^x) \cdot e(F(t,M), g_2^r) \\ &= e(h,y) \cdot e(F(t,M), \sigma'') \ . \end{aligned}$$

Note that $e(h,y)$ can be precomputed to save verification computation.

Given N signatures $\sigma_1, \ldots, \sigma_N \in \mathbb{G}_1 \times \mathbb{G}_2$ on the same message M at time t under N public keys $g_2^{x_1}, \ldots, g_2^{x_N}$, we can produce a multi-signature Σ on M by computing the coordinate-wise product of $\sigma_1, \ldots, \sigma_N$. Concretely, if $\sigma_i = (h^{x_i} \cdot F(t,M)^{r_i}, g_2^{r_i})$, then

$$\Sigma = (h^{x_1 + \cdots + x_N} \cdot F(t,M)^{r'}, g_2^{r'})$$

where $r' = r_1 + \cdots + r_N$. To verify Σ, we first compute a single *aggregate* public key that is a compressed version of all N individual public keys

$$apk \leftarrow y_1 \cdot \ldots \cdot y_N \ ,$$

and verify Σ against *apk* using the standard verification equation.

How to generate and update keys. To complete this overview, we describe a simplified version of the secret keys and update mechanism, where the secret keys are of size $O(T)$

instead of $O((\log T)^2)$. The construction exploits the fact that the function F satisfies

$$F(t,M) = F(t,0) \cdot F'^{M}$$

for some constant F'. This means that in order to sign messages at time t, it suffices to know

$$\widetilde{sk}_t = \{h^x \cdot F(t,0)^r, F'^r, g_2^r\}$$

from which we can compute $(h^x \cdot F(t,M)^r, g_2^r)$.

The secret key sk_t for time t is given by:

$$\widetilde{sk}_t, \widetilde{sk}_{t+1}, \cdots, \widetilde{sk}_T$$

generated using independent randomness. To update from the key sk_t to sk_{t+1}, we simply erase $\widetilde{sk}_t$. Forward security follows from the fact that an adversary who corrupts a signer at time t only learns sk_t and, in particular, does not learn $\widetilde{sk}_{t'}$ for $t' < t$, and is unable to create signatures for past time slots.

To compress the secret keys down to $O((\log T)^2)$ without increasing the signature size, we combine the tree-based approach in [23] with the compact HIBE in [15]. Roughly speaking, each sk_t now contains $\log T$ sub-keys, each of which contains $O(\log T)$ group elements and looks like an "expanded" version of $\widetilde{sk}_t$. (In the simplified scheme, each sk_t contains $T - t + 1$ sub-keys, each of which contains three group elements.)

Security against rogue-key attacks. The design of multi-signature schemes must take into account rogue-key attacks, where an adversary forges a multi-signature by providing specially crafted public keys that are correlated with the public keys of the honest parties. We achieve security against rogue-key attacks by having users provide a proof of possession of their secret key [14, 56]; it suffices here for each user to provide a standard BLS signature y' on its public key y (cf. the proof π in the key generation and verification algorithms in Section 5.2).

Avoiding trusted set-up. Note that the common parameters contain uniformly random group elements $h, h_0, \ldots, h_{\log T}$ in $\mathbb{G}_2$ which are used to define the function F. These elements can be generated using a indifferentiable hash-to-curve algorithm [20,63] evaluated on some fixed sequence of inputs (e.g. determined by the digits of pi), thereby avoiding any trusted set-up.

2.1 Discussion

Related works. The use of HIBE schemes for forward secrecy originates in the context of encryption [23] and has been used in signatures [19,27], key exchange [35] and proxy re-encryption [32]. Our signature scheme is quite similar to the forward-secure signatures of Boyen et al. [19] and achieves the same asymptotic complexity; their construction is more complex in order to achieve security against untrusted updates. The way we achieve aggregation is similar to the multi-signatures in [45].

Alternative approaches to posterior security. There are two variants of the posterior attack: (i) a short-range variant, where an adversary tries to corrupt a committee member prior to completion of the consensus sub-protocol, and (ii) a long-range variant as explained earlier. Dfinity [36], Ouroboros [41] and Casper [22] cope with the short-range attacks by assuming a delay in attacks that is longer than the running time of the consensus sub-protocol. For long-range attacks, Casper adopts a fork choice rule to never revert a finalized block, and in addition, assumes that clients log on with sufficient regularity to gain a complete update-to-date view of the chain. We note that forward-secure signatures provide a clean solution against both attacks, without the need for fork choice rules or additional assumptions about the adversary and the clients.

Application to permissioned blockchains. Consensus protocols, such as PBFT, are also at the core of many permissioned blockchains (e.g. Hyperledger), where only approved parties may join the network. Our signature scheme can similarly be applied to this setting to achieve forward secrecy, reduce communication bandwidth, and produce compact block certificates.

3 Preliminaries

Let $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t$ be multiplicative groups of prime order q with a non-degenerate pairing function $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_t$. Let g_1 and g_2 be generators of $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively.

In analogy with the weak bilinear Diffie-Hellman inversion problem ℓ-wBDHI* [15], which was originally defined for Type-1 pairings (i.e., symmetric pairings where we have $\mathbb{G}_1 = \mathbb{G}_2$), we define the following variant for Type-3 pairings denoted ℓ-wBDHI$^*_3$.

$$\begin{aligned}\text{Input: } & A_1 = g_1^{\alpha}, A_2 = g_1^{(\alpha^2)}, \ldots, A_\ell = g_1^{(\alpha^\ell)}, \\ & B_1 = g_2^{\alpha}, B_2 = g_2^{(\alpha^2)}, \ldots, B_\ell = g_2^{(\alpha^\ell)}, \\ & C_1 = g_1^{\gamma}, C_2 = g_2^{\gamma} \\ & \text{for } \alpha, \gamma \xleftarrow{\$} \mathbb{Z}_q \\ \text{Compute: } & e(g_1, g_2)^{(\gamma \cdot \alpha^{\ell+1})}\end{aligned}$$

The advantage $\mathbf{Adv}^{\ell\text{-wBDHI}^*_3}_{\mathbb{G}_1 \times \mathbb{G}_2}(\mathcal{A})$ of an adversary $\mathcal{A}$ is defined as its probability in solving this problem.

As shown in [15], the assumption holds in the generic bilinear group model, with a lower bound of $\Omega(\sqrt{q/\ell})$ (with a matching attack in [26]). Concretely, for the BLS12-381

pairing-friendly curve with $\ell = 32$, the best attack has complexity roughly 2^{125}.

Alternatively, our scheme could be proved secure under a variant of the above assumption where the adversary has to output $g_1^{(\alpha^{\ell+1})}$ given as input $A_1,\ldots,A_\ell,B_1,\ldots,B_\ell$ and given access to an oracle $\psi : g_2^x \mapsto g_1^x$. Because of the ψ oracle, this assumption is incomparable to the ℓ-wBDHI assumption described above.

4 Forward-Secure Signatures

We begin by describing a forward-secure signature scheme, and then extend the construction to a multi-signature scheme in Section 5.

4.1 Definition

We use the Bellare-Miner model [9] to define syntax and security of a forward-secure signature scheme. A forward-secure signature scheme $\mathcal{FS}$ for a message space $\mathcal{M}$ consists of the following algorithms:

Setup: $pp \xleftarrow{\$} \mathsf{Setup}(T)$. All parties agree on the public parameters pp. The setup algorithm mainly fixes the distribution of the parameters given the maximum number of time periods T. The parameters may be generated by a trusted third party, through a distributed protocol, or set to "nothing-up-my-sleeve" numbers. The public parameters are taken to be an implicit input to all of the following algorithms.

Key generation: $(pk, sk_1) \xleftarrow{\$} \mathsf{Kg}$. The signer runs the key generation algorithm on input the maximum number of time periods T to generate a public verification key pk and an initial secret signing key sk_1 for the first time period.

Key update: $sk_{t+1} \xleftarrow{\$} \mathsf{Upd}(sk_t)$. The signer updates its secret key sk_t for time period t to sk_{t+1} for the next period using the key update algorithm. The scheme could also offer a "fast-forward" update algorithm $sk_{t'} \xleftarrow{\$} \mathsf{Upd}'(sk_t, t')$ for any $t' > t$ that is more efficient than repetitively applying Upd.

Signing: $\sigma \xleftarrow{\$} \mathsf{Sign}(sk_t, M)$. On input the current signing key sk_t and message $M \in \mathcal{M}$, the signer uses this algorithm to compute a signature σ.

Verification. $b \leftarrow \mathsf{Vf}(pk, t, M, \sigma)$. Anyone can verify a signature σ for on message M for time period t under public key pk by running the verification algorithm, which returns 1 to indicate that the signature is valid and 0 otherwise.

Correctness.

Correctness requires that for all messages $M \in \mathcal{M}$ and for all time periods $t \in [T]$ it holds that

$$\Pr[\mathsf{Vf}(pk, t, M, \mathsf{Sign}(sk_t, M)) = 1] = 1$$

where the coin tosses are over $pp \xleftarrow{\$} \mathsf{Setup}(T)$, $(pk, sk_1) \xleftarrow{\$} \mathsf{Kg}$, and $sk_i \leftarrow \mathsf{Upd}(sk_{i-1})$ for $i = 2,\ldots,t$.

Moreover, if the scheme has a fast-forward update algorithm, then the keys it produces must be distributed identically to those produced by repetitive application of the regular update algorithm. Meaning, for all $t, t' \in [T]$ with $t < t' \leq T$ and for all sk_t it holds that $sk'_{t'} \xleftarrow{\$} \mathsf{Upd}'(sk_t, t')$ follows the same distribution as sk_t produced as $sk_i \xleftarrow{\$} \mathsf{Upd}(sk_{i-1})$ for $i = t+1,\ldots,t'$.

Security.

Unforgeability under chosen-message attack for forward-secure signatures is defined through the following game. The experiment generates a fresh key pair (pk, sk_1) and hands the public key pk to the adversary $\mathcal{A}$. The adversary is given access to the following oracles:

Key update. If the current time period t (initially set to $t = 1$) is less than T, then this oracle updates the key sk_t to sk_{t+1} and increases t.

Signing. On input a message M, this oracle runs the signing oracle with the current secret key sk_t and message M, and returns the resulting signature σ.

Break in. The experiment records the break-in time $\bar{t} \leftarrow t$ and hands the current signing key $sk_{\bar{t}}$ to the adversary. This oracle can only be queried once, and after it has been queried, the adversary can make no further queries to the key update or signing oracles.

At the end of the game, the adversary outputs its forgery (t^*, M^*, σ^*). It wins the game if σ^* verifies correctly under pk for time period t^* and message M^*, if it never queried the signing oracle on M^* during time period t^*, and if it queried the break-in oracle, then it did so in a time period $\bar{t} > t^*$. We define $\mathcal{A}$'s advantage $\mathbf{Adv}^{\text{fu-cma}}_{\mathcal{FS}}(\mathcal{A})$ as its probability in winning the above game.

We also define a selective variant of the above notion, referred to as sfu-cma, where the adversary first has to commit to $\bar{t}$, t^*, and M^*. More specifically, $\mathcal{A}$ first outputs $(\bar{t}, t^*, M^*)$, then receives the public key pk, is allowed to make signature and key update queries until time period $t = \bar{t}$ is reached, at which point it is given $sk_{\bar{t}}$ and outputs its forgery σ^*.

4.2 Encoding time periods

Following [23], we associate time periods with all nodes of the tree according to a pre-order traversal. Prior tree-based forward-secure signatures [9, 48] associate time periods with the only leaf nodes; using all nodes allows us to reduce the amortized complexity of key updates from $O(\log T)$ exponentiations to $O(1)$ exponentiations.

Recall that a tree of depth $\ell - 1$ has $2^\ell - 1$ nodes, which then correspond to time periods in $[2^\ell - 1]$. We will identify

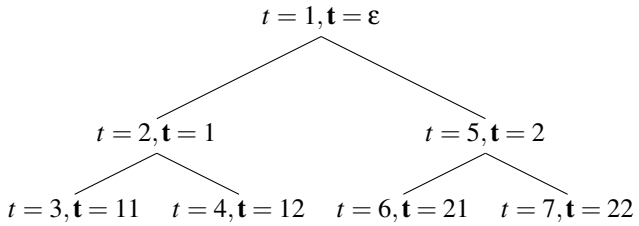

Figure 2: Tree structure illustrating bijection between $t \in [2^\ell - 1]$ and $\mathbf{t} \in \{1,2\}^{\leq \ell-1}$ for $\ell = 3$.

the nodes of the tree of depth $\ell-1$ with strings in $\{1,2\}^{\leq \ell-1}$ where 1 denotes taking the left branch and 2 denotes taking the right branch. An example of such a tree is depicted in Figure 2. We work with $\{1,2\}$ instead of $\{0,1\}$ for technical reasons: roughly speaking, in the scheme, we need to work with strings of length exactly $\ell-1$, which we obtain by padding strings in $\{1,2\}^{\leq \ell-1}$ with zeroes.

We can also describe the association explicitly as a bijection between $\mathbf{t} = t_1 \| t_2 \| \ldots \in \{1,2\}^{\leq \ell-1}$ and $t \in [2^\ell - 1]$ for any integer ℓ given by

$$t(\mathbf{t}) = 1 + \sum_{i=1}^{|\mathbf{t}|} (1 + 2^{\ell-i}(t_i - 1)) \; .$$

For instance, for $\ell = 3$, this maps $\varepsilon, 1, 11, 12, 2, 21, 22$ to $1, 2, 3, 4, 5, 6, 7$. The inverse of the bijection can be described as

$$\begin{array}{lll} \mathbf{t}(1) & = \varepsilon & \\ \mathbf{t}(t) & = \mathbf{t}(t-1) \| 1 & \text{if } |\mathbf{t}(t-1)| < \ell - 1 \\ \mathbf{t}(t) & = \bar{\mathbf{t}} \| 2 & \text{if } |\mathbf{t}(t-1)| = \ell - 1 \end{array}$$

where $\bar{\mathbf{t}}$ is the longest string such that $\bar{\mathbf{t}} \| 1$ is a prefix of $\mathbf{t}(t-1)$.

The bijection induces a natural precedence relation over $\{1,2\}^{\leq \ell-1}$ where $\mathbf{t} \preceq \mathbf{t}'$ iff either $\mathbf{t}$ is a prefix of $\mathbf{t}'$ or exists $\bar{\mathbf{t}}$ s.t. $\bar{\mathbf{t}} \| 1$ is a prefix of $\mathbf{t}$ and $\bar{\mathbf{t}} \| 2$ is a prefix of $\mathbf{t}'$. We also write $\mathbf{t}, \mathbf{t}+1$ corresponding to $t, t+1$.

Next, we associate any $\mathbf{t} \in \{1,2\}^{\leq \ell-1}$ with a set $\Gamma_{\mathbf{t}} \subset \{1,2\}^{\leq \ell-1}$ given by

$$\Gamma_{\mathbf{t}} := \{ \mathbf{t} \} \cup \{ \bar{\mathbf{t}} \| 2 : \bar{\mathbf{t}} \| 1 \text{ prefix of } \mathbf{t} \}$$

that corresponds to the set containing $\mathbf{t}$ and all the right-hand siblings of nodes on the path from $\mathbf{t}$ to the root, which also happens to be the smallest set of nodes that includes a prefix of all $\mathbf{t}' \succeq \mathbf{t}$. For instance, for $\ell = 3$, we have

$$\Gamma_1 = \{1,2\}, \Gamma_{11} = \{11, 12, 2\}, \Gamma_{12} = \{12, 2\} \; .$$

The sets $\Gamma_{\mathbf{t}}$ satisfy the following properties:

- $\mathbf{t}' \succeq \mathbf{t}$ iff there exists $\mathbf{u} \in \Gamma_{\mathbf{t}}$ s.t. $\mathbf{u}$ is a prefix of $\mathbf{t}'$;
- For all $\mathbf{t}$, we have $\Gamma_{\mathbf{t}+1} = \Gamma_{\mathbf{t}} \setminus \{\mathbf{t}\}$ if $|\mathbf{t}| = \ell - 1$ or $\Gamma_{\mathbf{t}+1} = (\Gamma_{\mathbf{t}} \setminus \{\mathbf{t}\}) \cup \{\mathbf{t}\|1, \mathbf{t}\|2\}$ otherwise;
- For all $\mathbf{t}' \succ \mathbf{t}$, we have that for all $\mathbf{u}' \in \Gamma_{\mathbf{t}'}$, there exists $\mathbf{u} \in \Gamma_{\mathbf{t}}$ such that $\mathbf{u}$ is a prefix of $\mathbf{u}'$.

The first property is used for verification and for reasoning about security; the second and third properties are used for key updates.

4.3 Construction

We assume the bound T is of the form $2^\ell - 1$. We use the above bijection so that the algorithms take input $\mathbf{t} \in \{1,2\}^{\leq \ell-1}$ instead of $t \in [T]$. The following scheme is roughly the result of applying the Canetti-Halevi-Katz technique to obtain forward security from hierarchical identity-based encryption (HIBE) [24] to the signature scheme determined by the key structure of the Boneh-Boyen-Goh HIBE scheme [15]; we describe the differences at the end of this subsection.

Setup. Let $\mathcal{M}$ be the message space of the scheme and let $\mathsf{H}_q : \mathcal{M} \to \{0,1\}^\kappa$ be a hash function that maps messages to bit strings of length κ such that $2^\kappa < q$. Apart from the description of the groups, the common system parameters also contain the maximum number of time slots $T = 2^\ell - 1$ and random group elements $h, h_0, \ldots, h_\ell \xleftarrow{\$} \mathbb{G}_1$. These parameters could, for example, be generated as the output of a hash function modeled as a random oracle.

Key generation. Each signer chooses $x \xleftarrow{\$} \mathbb{Z}_q$ and computes $y \leftarrow g_2^x$. It sets its public to $pk = y$ and computes its initial secret key $sk_1 \leftarrow \{\widetilde{sk}_\varepsilon\}$ where $\widetilde{sk}_\varepsilon = \left(g_2^r \, , \, h^x h_0^r \, , \, h_1^r, \ldots, h_\ell^r\right)$ for $r \xleftarrow{\$} \mathbb{Z}_q$.

Key update. We associate with each $\mathbf{w} \in \{1,2\}^k$ with $k \leq \ell - 1$ a key $\widetilde{sk}_{\mathbf{w}}$ of the form

$$\begin{array}{rcl} \widetilde{sk}_{\mathbf{w}} & = & (c, d, e_{k+1}, \ldots, e_\ell) \\ & = & \left(g_2^r \, , \, h^x (h_0 \prod_{j=1}^{k} h_j^{w_j})^r \, , \, h_{k+1}^r \, , \, \ldots \, , \, h_\ell^r \right) \quad (1) \end{array}$$

for $r \xleftarrow{\$} \mathbb{Z}_q$. Given $\widetilde{sk}_{\mathbf{w}}$, one can derive a key for any $\mathbf{w}' \in \{1,2\}^{k'}$ which contains $\mathbf{w}$ as a prefix as

$$(c', d', e'_{k'+1}, \ldots, e'_\ell) = \left(c \cdot g_2^{r'} \, , \, d \cdot \prod_{j=k+1}^{k'} e_j^{w_j} \cdot (h_0 \prod_{j=1}^{k'} h_j^{w_j})^{r'} \, , \right.$$
$$\left. e_{k'+1} \cdot h_{k'+1}^{r'} \, , \, \ldots \, , \, e_\ell \cdot h_\ell^{r'} \right) \quad (2)$$

for $r' \xleftarrow{\$} \mathbb{Z}_q$.

The secret key $sk_{\mathbf{t}}$ at time period $\mathbf{t}$ is given by

$$sk_{\mathbf{t}} = \{ \widetilde{sk}_{\mathbf{w}} : \mathbf{w} \in \Gamma_{\mathbf{t}} \} \; ,$$

which, by the first property of $\Gamma_{\mathbf{t}}$, contains a key $\widetilde{sk}_{\mathbf{w}}$ for a prefix $\mathbf{w}$ of all nodes $\mathbf{t}' \succeq \mathbf{t}$.

To perform a regular update of $sk_{\mathbf{t}}$ to $sk_{\mathbf{t}+1}$, the signer users the second property of $\Gamma_{\mathbf{t}}$. Namely, if $|\mathbf{t}| < \ell - 1$, then the signer looks up $\widetilde{sk}_{\mathbf{t}} = (c, d, e_{|\mathbf{t}|+1}, \ldots, e_\ell) \in sk_{\mathbf{t}}$, computes

$$\widetilde{sk}_{\mathbf{t}\|1} \leftarrow (c, d \cdot e_{|\mathbf{t}|+1}, e_{|\mathbf{t}|+2}, \ldots, e_\ell) \ ,$$

and derives $\widetilde{sk}_{\mathbf{t}\|2}$ from $\widetilde{sk}_{\mathbf{t}}$ using Equation (2). The signer then sets $sk_{\mathbf{t}+1} \leftarrow (sk_{\mathbf{t}} \setminus \widetilde{sk}_{\mathbf{t}}) \cup \{\widetilde{sk}_{\mathbf{t}\|1}, \widetilde{sk}_{\mathbf{t}\|2}\}$ and securely deletes $sk_{\mathbf{t}}$ as well as the re-randomization exponent r' used in the derivation of $\widetilde{sk}_{\mathbf{t}\|2}$.

If $|\mathbf{t}| = \ell - 1$, then the signer simply sets $sk_{\mathbf{t}+1} \leftarrow sk_{\mathbf{t}} \setminus \{\widetilde{sk}_{\mathbf{t}}\}$ and securely deletes $sk_{\mathbf{t}}$.

To perform a fast-forward update of its key to any time $\mathbf{t}' \succeq \mathbf{t}$, the signer derives keys $\widetilde{sk}_{\mathbf{w}'}$ for all nodes $\mathbf{w}' \in \Gamma_{\mathbf{t}'} \setminus \Gamma_{\mathbf{t}}$ by applying Equation (2) to the key $\widetilde{sk}_{\mathbf{w}} \in sk_{\mathbf{t}}$ such that $\mathbf{w}$ is a prefix of $\mathbf{w}'$, which must exist due to the third property of $\Gamma_{\mathbf{t}}$. The signer then sets $sk_{\mathbf{t}'} \leftarrow \{\widetilde{sk}_{\mathbf{w}'} : \mathbf{w}' \in \Gamma_{\mathbf{t}'}\}$ and securely deletes $sk_{\mathbf{t}}$ as well as all re-randomization exponents used in the key derivations.

Signing. To generate a signature on message $M \in \mathcal{M}$ in time period $\mathbf{t} \in \{1,2\}^{\leq \ell-1}$, the signer looks up $\widetilde{sk}_{\mathbf{t}} = (c, d, e_{|\mathbf{t}|+1}, \ldots, e_\ell) \in sk_{\mathbf{t}}$, chooses $r' \xleftarrow{\$} \mathbb{Z}_q$, and outputs

$$(\sigma_1, \sigma_2) \ = \ \left(d \cdot e_\ell^{\mathsf{H}_q(M)} \cdot \big(h_0 \cdot \prod_{j=1}^{|\mathbf{t}|} h_j^{t_j} \cdot h_\ell^{\mathsf{H}_q(M)}\big)^{r'} \ , \ c \cdot g_2^{r'} \right) .$$

Verification. Anyone can verify a signature $(\sigma_1, \sigma_2) \in \mathbb{G}_1 \times \mathbb{G}_2$ on message M under public key $pk = y$ in time period $\mathbf{t}$ by checking whether

$$\mathrm{e}(\sigma_1, g_2) \ = \ \mathrm{e}(h, y) \cdot \mathrm{e}\big(h_0 \cdot \prod_{j=1}^{|\mathbf{t}|} h_j^{t_j} \cdot h_\ell^{\mathsf{H}_q(M)} \ , \ \sigma_2\big) \ .$$

Note that the pairing $\mathrm{e}(h, y)$ can be pre-computed from the public key ahead of time, so that verification only requires two pairing computations.

Differences from prior works. We highlight the differences between our scheme and those in [15, 19, 23], assuming some familiarity with these prior constructions.

- We rely on asymmetric bilinear groups for efficiency, and our signature sits in $\mathbb{G}_2 \times \mathbb{G}_1$ instead of $\mathbb{G}_2^2$. This way, it is sufficient to give out the public parameters $h_0, \ldots, h_\ell$ in $\mathbb{G}_1$ (which we can then instantiate using hash-to-curve without trusted set-up) instead of having to generate "consistent" public parameters $(h_i, h_i') = (g_1^{x_i}, g_2^{x_i}) \in \mathbb{G}_1 \times \mathbb{G}_2$.
- Our key-generation algorithm also deviates from that in the Boneh-Boyen-Goh HIBE, which would set

$$pk = \mathrm{e}(g_1, g_2)^x, h = g_1, \widetilde{sk}_\varepsilon = \big(g_2^r, g_1^x h_0^r, h_1^r, \ldots, h_\ell^r\big).$$

  In our scheme, $pk = g_2^x$ lies in $\mathbb{G}_2$ instead of $\mathbb{G}_t$ and is therefore smaller. Setting h to be random instead of g_1 also allows us to achieve security under weaker assumptions. In fact, setting $h = g_1$ and $pk = g_2^x$ would yield an insecure scheme in symmetric pairing groups where $g_1 = g_2$, since $h^x = g_1^x = g_2^x = pk$.

4.4 Correctness

We say that a secret key $sk_{\mathbf{t}}$ for time period $\mathbf{t}$ is *well-formed* if $sk_{\mathbf{t}} = \{\widetilde{sk}_{\mathbf{w}} : \mathbf{w} \in \Gamma_{\mathbf{t}}\}$, where each $\widetilde{sk}_{\mathbf{w}}$ is of the form of Equation (1) for an independent uniformly distributed exponent $r \xleftarrow{\$} \mathbb{Z}_q$. We first show that all honestly generated and updated secret keys are well-formed, and then proceed to the verification of signatures.

The key $sk_{\mathbf{t}}$ is trivially well-formed for $t = 1$, i.e., $\mathbf{t} = \varepsilon$, as can be seen from the key generation algorithm. We now show that $sk_{\mathbf{t}}$ is also well-formed after a regular update from time $\mathbf{t}$ to $\mathbf{t}+1$ and after a fast-forward update from $\mathbf{t}$ to $\mathbf{t}' \succ \mathbf{t}$.

In a regular update, assume that $sk_{\mathbf{t}}$ is well-formed. If $|\mathbf{t}| = \ell - 1$, then the update procedure sets $sk_{\mathbf{t}+1} \leftarrow sk_{\mathbf{t}} \setminus \{\widetilde{sk}_{\mathbf{t}}\}$, which by the second property of $\Gamma_{\mathbf{t}}$ and the induction hypothesis means that $sk_{\mathbf{t}+1}$ is also well-formed. If $|\mathbf{t}| < \ell - 1$, the update procedure adds keys $\widetilde{sk}_{\mathbf{t}\|1}$ and $\widetilde{sk}_{\mathbf{t}\|2}$ and removes $\widetilde{sk}_{\mathbf{t}}$ from $sk_{\mathbf{t}}$, which by the second property of $\Gamma_{\mathbf{t}}$ indeed corresponds to $\{\mathbf{w} : \mathbf{w} \in \Gamma_{\mathbf{t}+1}\}$. Moreover, $\widetilde{sk}_{\mathbf{t}\|1}$ is derived from $\widetilde{sk}_{\mathbf{t}} = \widetilde{sk}_{\mathbf{t}\|1} \leftarrow (c, d, e_{|\mathbf{t}|+1}, \ldots, e_\ell)$ as $\widetilde{sk}_{\mathbf{t}\|1} \leftarrow (c, d \cdot e_{|\mathbf{t}|+1}, e_{|\mathbf{t}|+2}, \ldots, e_\ell)$, which satisfies Equation (1) with randomness r that is independent from all other keys in $sk_{\mathbf{t}+1}$ because $\widetilde{sk}_{\mathbf{t}} \notin sk_{\mathbf{t}+1}$. Similarly, $\widetilde{sk}_{\mathbf{t}\|2}$ satisfies Equation (1) because it is generated as

$$\begin{aligned}
c' \ &= \ c \cdot g_2^{r'} \ = \ g_2^{r+r'} \\
d' \ &= \ d \cdot e_{k+1} \cdot (h_0 \prod_{j=1}^{k} h_j^{t_j} \cdot h_{k+1}^{w_{k+1}})^{r'} \\
&= \ h^x (h_0 \prod_{j=1}^{k} h_j^{t_j} \cdot h_{k+1}^2)^{r+r'} \\
e'_{k+2} \ &= \ e_{k+2} \cdot h_{k+2}^{r'} \ = \ h_{k+2}^{r+r'} \\
&\vdots \\
e'_\ell \ &= \ e_\ell \cdot h_\ell^{r'} \ = \ h_\ell^{r+r'}
\end{aligned}$$

satisfying Equation (1) with randomness $r + r'$, which is independent of the randomness of other keys in $sk_{\mathbf{t}+1}$ due to the uniform choice of r'.

For the fast-forward update procedure, one can see that if $sk_{\mathbf{t}}$ is well-formed, then the updated key $sk_{\mathbf{t}'}$ for $\mathbf{t}' \succ \mathbf{t}$ is well-formed as well. Indeed, by adding the keys for nodes in $\Gamma_{\mathbf{t}'} \setminus \Gamma_{\mathbf{t}}$ and removing those for $\Gamma_{\mathbf{t}} \setminus \Gamma_{\mathbf{t}'}$, we have that $sk_{\mathbf{t}'}$ contains keys $\widetilde{sk}_{\mathbf{w}}$ for all $\mathbf{w} \in \Gamma_{\mathbf{t}'}$. The randomness independence is guaranteed by the random choice of r' in Equation (2). In the

optimized variant, all keys still have independent randomness because one key $\widetilde{sk}_{\mathbf{w}'} \in sk_{\mathbf{t}'}$ will have the same randomness r as some key $\widetilde{sk}_{\mathbf{w}} \in sk_{\mathbf{t}}$ where $\mathbf{w}$ is a prefix of $\mathbf{w}'$. That randomness is independent from all other keys in $sk_{\mathbf{t}'}$, however, because the key $\widetilde{sk}_{\mathbf{w}}$ does not occur in $sk_{\mathbf{t}'}$. Indeed, by the definition of $\Gamma_{\mathbf{t}'}$, one can see that $\Gamma_{\mathbf{t}'}$ cannot have elements $\mathbf{w} \neq \mathbf{w}'$ with $\mathbf{w}$ a prefix of $\mathbf{w}'$.

To see why signature verification works, observe that a signature for time period $\mathbf{t}$ and message M is computed from a key $\widetilde{sk}_{\mathbf{t}} = (c, d, e_{|\mathbf{t}|+1}, \ldots, e_\ell)$ in a well-formed key $sk_{\mathbf{t}}$. The left-hand side of the verification equation is therefore

$$\begin{aligned}
\mathrm{e}(\sigma_1, g_2) &= \mathrm{e}\left(d \cdot e_\ell^{\mathsf{H}_q(M)} \cdot \big(h_0 \cdot \prod_{j=1}^{|\mathbf{t}|} h_j^{t_j} \cdot h_\ell^{\mathsf{H}_q(M)}\big)^{r'} \,,\, g_2\right) \\
&= \mathrm{e}\left(h^x \big(h_0 \cdot \prod_{j=1}^{|\mathbf{t}|} h_j^{t_j} \cdot h_\ell^{\mathsf{H}_q(M)}\big)^{r+r'} \,,\, g_2\right) \\
&= \mathrm{e}(h^x, g_2) \cdot \mathrm{e}\left(h_0 \cdot \prod_{j=1}^{|\mathbf{t}|} h_j^{t_j} \cdot h_\ell^{\mathsf{H}_q(M)} \,,\, g_2\right)^{r+r'} \\
&= \mathrm{e}(h, y) \cdot \mathrm{e}\left(h_0 \cdot \prod_{j=1}^{|\mathbf{t}|} h_j^{t_j} \cdot h_\ell^{\mathsf{H}_q(M)} \,,\, \sigma_2\right) .
\end{aligned}$$

4.5 Security

Theorem 1. *For any* fu-cma *adversary $\mathcal{A}$ against the above forward-secure signature scheme in the random-oracle model for $T = 2^\ell - 1$ time periods, there exists an adversary $\mathcal{B}$ with essentially the same running time and advantage in solving the ℓ-wBDHI$^*_3$ problem*

$$\mathbf{Adv}^{\ell\text{-wBDHI}^*_3}_{\mathbb{G}_1 \times \mathbb{G}_2}(\mathcal{B}) \;\geq\; \frac{1}{T \cdot q_{\mathrm{H}}} \cdot \mathbf{Adv}^{\text{fu-cma}}_{\mathcal{FS}}(\mathcal{A}) - \frac{q_{\mathrm{H}}^2}{2^\kappa} \,,$$

where q_{H} is the number of random-oracle queries made by $\mathcal{A}$.

We refer the interested reader to Appendix A for the full proof of security.

5 Forward-Secure Multi-Signatures

To obtain a multi-signature scheme, we observe that the component-wise product $(\Sigma_1, \Sigma_2) = (\prod_{i=1}^n \sigma_{i,1}, \prod_{i=1}^n \sigma_{i,2})$ of a number of signatures $(\sigma_{1,1}, \sigma_{1,2}), \ldots, (\sigma_{n,1}, \sigma_{n,2})$ satisfies the verification equation with respect of the product of public keys $Y = y_1 \cdot \ldots \cdot y_n$. This method of combining signatures is vulnerable to a rogue-key attack, however, where a malicious signer chooses his public key based on that of an honest signer, so that the malicious signer can compute valid signatures for their aggregated public key. The scheme below borrows a technique due to Ristenpart and Yilek [56] using proofs of possession (denote by π below) to prevent against these types of attack.

5.1 Definitions

In addition to the algorithms of a forward-secure signature scheme in Section 4.1, a forward-secure multi-signature scheme $\mathcal{FMS}$ in the key verification model has a key generation that additionally outputs a proof π for the public key:

Key generation: $(pk, \pi, sk_1) \xleftarrow{\$} \mathsf{Kg}$. The key generation algorithm generates a public verification key pk, a proof π, and an initial secret signing key sk_1 for the first time period.

and additionally has the following algorithms:

Key verification: $b \leftarrow \mathsf{KVf}(pk, \pi)$. The key verification algorithm returns 1 if the proof pk is valid for pk and returns 0 otherwise.

Key aggregation: $apk \xleftarrow{\$} \mathsf{KAgg}(pk_1, \ldots, pk_n)$. On input a list of individual public keys $(pk_1, \ldots, pk_n)$, the key aggregation returns an aggregate public key apk, or $\perp$ to indicate that key aggregation failed.

Signature aggregation. $\Sigma \xleftarrow{\$} \mathsf{SAgg}((pk_1, \sigma_1), \ldots, (pk_n, \sigma_n), t, M)$. Anyone can aggregate a given list of individual signatures $(\sigma_1, \ldots, \sigma_n)$ by different signers with public keys $(pk_1, \ldots, pk_n)$ on the same message M and for the same period t into a single multi-signature Σ.

Aggregate verification. $b \leftarrow \mathsf{AVf}(apk, t, M, \Sigma)$. Given an aggregate public key apk, a message M, a time period t, and a multi-signature Σ, the verification algorithm returns 1 to indicate that all signers in apk signed M in period t, or 0 to indicate that verification failed.

Correctness. Correctness requires that $\mathsf{KVf}(pk, \pi) = 1$ with probability one if $(pk, \pi, sk_1) \xleftarrow{\$} \mathsf{Kg}$ and that for all messages $M \in \mathcal{M}$, for all $n \in \mathbb{Z}$, and for all time periods $t \in \{0, \ldots, T-1\}$, it holds that $\mathsf{AVf}(apk, t, M, \Sigma) = 1$ with probability one if $(pk_i, \pi_i, sk_{i,1}) \xleftarrow{\$} \mathsf{Kg}$, $apk \xleftarrow{\$} \mathsf{KAgg}(pk_1, \ldots, pk_n)$, $sk_{i,j} \xleftarrow{\$} \mathsf{Upd}(sk_{i,j-1})$ for $i = 1, \ldots, n$ and $j = 2, \ldots, t$, $\sigma_i \xleftarrow{\$} \mathsf{Sign}(sk_{i,t}, M)$ for $i = 1, \ldots, n$, and $\Sigma \xleftarrow{\$} \mathsf{SAgg}((pk_1, \sigma_1), \ldots, (pk_n, \sigma_n), t, M)$.

Security. Unforgeability (fu-cma) is defined through a game that is similar to that described in Section 4.1. The adversary is given the public key pk and proof π of an honest signer and access to the same key update, signing, and break-in oracles. However, at the end of the game, the adversary's forgery consists of a list of public keys and proofs $(pk_1^*, \pi_1^*, \ldots, pk_n^*, \pi_n^*)$, a message M^*, a time period t^*, and a multi-signature Σ^*. The forgery is considered valid if

- $pk \in \{pk_1^*, \ldots, pk_n^*\}$,
- the proofs $\pi_1^*, \ldots, \pi_n^*$ are valid for public keys $pk_1^*, \ldots, pk_n^*$ according to KVf,

- Σ^* is valid with respect to the aggregate public key apk^* of $(pk_1^*,\ldots,pk_n^*)$, message M^*, and time period t^*,
- $\bar{t} > t^*$,
- and $\mathcal{A}$ never made a signing query for M^* during time period t^*.

Our security model covers rogue-key attacks because the adversary first receives the target public key pk, and only then outputs the list of public keys $pk_1^*,...,pk_n^*$ involved in its forgery. The only condition on these public keys is that they are accompanied by valid proofs $\pi_1^*,...,\pi_n^*$.

5.2 Construction

Let $\mathsf{H}_{\mathbb{G}_1} : \{0,1\}^* \to \mathbb{G}_1^*$ be a hash function. The multi-signature scheme reuses the key update and signature algorithms from the scheme from Section 4.3, but uses different key generation and verification algorithms, and adds signature and key aggregation.

Key generation. Each signer chooses $x \xleftarrow{\$} \mathbb{Z}_q$ and computes $y \leftarrow g_2^x$ and $y' \leftarrow \mathsf{H}_{\mathbb{G}_1}(\mathtt{PoP}, y)$, where $\mathtt{PoP}$ is a fixed string used as a prefix for domain separation. It sets its public key to $pk = y$, the proof to $\pi = y'$, and computes its initial secret key as $sk_1 \leftarrow h^x$.

Key verification. Given a public key $pk = y$ with proof $\pi = y'$, the key verification algorithm validates the proof of possession by returning 1 if

$$\mathrm{e}(y', g_2) \;=\; \mathrm{e}(\mathsf{H}_{\mathbb{G}_1}(\mathtt{PoP}, y), y)$$

and returning 0 otherwise.

Key aggregation. Given public keys $pk_1 = y_1,\ldots,$ $pk_n = y_n$, the key aggregation algorithm computes $Y \leftarrow \prod_{i=1}^n y_i$ and returns the aggregate public key $apk = Y$.

Signature aggregation. Given signatures $\sigma_1 = (\sigma_{1,1},\sigma_{1,2}),\ldots,\sigma_n = (\sigma_{n,1},\sigma_{n,2}) \in \mathbb{G}_1 \times \mathbb{G}_2$ on the same message M, the signature aggregation algorithm outputs

$$\Sigma \;=\; (\Sigma_1,\Sigma_2) \;=\; \Big(\prod_{i=1}^n \sigma_{i,1}\;,\; \prod_{i=1}^n \sigma_{i,2}\Big)\;.$$

Aggregate verification. Multi-signatures are verified with respect to aggregate public keys in exactly the same way as individual signatures with respect to individual public keys. Namely, given a multi-signature $(\Sigma_1,\Sigma_2) \in \mathbb{G}_1 \times \mathbb{G}_2$ on message M under aggregate public key $apk = Y$ in time period $\mathbf{t}$, the verifier accepts if and only if $apk \neq \perp$ and

$$\mathrm{e}(\Sigma_1, g_2) \;=\; \mathrm{e}(h, Y)\cdot \mathrm{e}\Big(h_0 \cdot \prod_{j=1}^{|\mathbf{t}|} h_j^{t_j} \cdot h_{\ell+1}^{\mathsf{H}_q(M)}\;,\; \Sigma_2\Big)\;.$$

5.3 Security

Theorem 2. *For any* fu-cma *adversary $\mathcal{A}$ against the above forward-secure multi-signature scheme for $T = 2^\ell - 1$ time periods in the random-oracle model, there exists an adversary $\mathcal{B}$ with essentially the same running time that solves the ℓ-wBDHI$_3^*$ problem with advantage*

$$\mathbf{Adv}^{\ell\text{-wBDHI}_3^*}_{\mathbb{G}_1\times\mathbb{G}_2}(\mathcal{B}) \;\geq\; \frac{1}{T\cdot q_{\mathrm{H}}}\cdot \mathbf{Adv}^{\text{fu-cma}}_{\mathcal{FMS}}(\mathcal{A}) - \frac{q_{\mathrm{H}}^2}{2^\kappa}\;,$$

where q_{H} is the number of random-oracle queries made by $\mathcal{A}$.

We prove the theorem by showing that a forger $\mathcal{A}$ for the multi-signature scheme yields a forger $\mathcal{A}'$ for the single-signer scheme of Section 4.3 such that $\mathbf{Adv}^{\text{fu-cma}}_{\mathcal{FS}}(\mathcal{A}') \geq \mathbf{Adv}^{\text{fu-cma}}_{\mathcal{FS}}(\mathcal{A})$. The theorem then follows from Theorem 1.

The key idea for the proof of security following [56] is to program $\mathsf{H}_{\mathbb{G}_1}$ in such a way that we can "extract" a valid forgery for the single-signer scheme starting from that for the multi-signature scheme. In particular,

- given a rogue public key $pk_i^* = y_i$ with proof $\pi_i^* = y_i'$ where $y_i = g_2^{x_i}$, we can extract the corresponding secret key h^{x_i} from y_i' by programming $\mathsf{H}_{\mathbb{G}_1}(\mathtt{PoP}, y_i) = h^{r_i}$.
- given h^{x_i} for all $y_i \neq y$ along with a valid forgery for the multi-signature scheme, we can extract a forgery for the single-signer scheme. Here, we use the proofs $\pi_i^* = (h_i^x)_i^r$ to extract h_i^x for all adversarial keys $pk_i^* = g_2^{x_i}$.

We defer the interested reader to Appendix B for proof details.

6 Pixel in PoS-based Blockchains

In this section, we describe how to integrate Pixel into PoS-based blockchains that rely on forward-secure signatures to achieve security against posterior corruptions. We summarize systems that rely on forward-secure signatures, abstract how signatures are used in these systems, and explain how to apply Pixel.

PoS Blockchains Secure under Posterior Corruptions. Ouroboros Genesis and Praos rely on forward-secure signatures to protect against posterior corruptions [5,31,40]. These blockchains require users to rotate key and assume secure erasures. Thuderella is a blockchain with fast optimistic instant confirmation [54]. The blockchain is secure against posterior corruptions assuming that a majority of the computing power is controlled by honest players. Similarly, the protocol relies on forward-secure signatures. Pixel can be applied in all these blockchains to protect against posterior attacks and potentially reduce bandwidth, storage, and computation costs in instances where many users propagate many signatures on

the same message (e.g., a block of transactions). Ouroboros Crypsinous uses forward-secure encryption to protect against the same attack [39]. Snow White shows that under a mild setup assumption, when nodes join the system they can access a set of online nodes the majority of whom are honest, the system can defend against posterior corruption attack [30]. The system does not rely on forward-secure signatures.

Background on PoS Blockchains. A blockchain is an append-only public ledger to which anyone can write and read. The fundamental problem in blockchains is to agree on a block of transactions between users. In Proof-of-Stake protocols, users map the stake or tokens they own in the system to "voting power" in the agreement protocol. Various types of PoS systems exist that use different formulas for determining the weight of each vote. For instance, in bounded PoS protocols, users must explicitly lock some amount A of their tokens to participate in the agreement. The weight of each vote is A/Q, where Q is the total number of locked tokens who's users wish to participate in the agreement. Users that misbehave are punished by a penalty applied to their locked tokens.

To tolerate malicious users, all PoS protocols run a Byzantine sub-protocol to agree on a block of transactions. The system is secure, assuming that that majority (often $2/3$) of the tokens participating in the consensus is honest. Each block is valid if a majority of committee members, weighted by their stake, approved it.

Pixel **Integration.** In order to vote on a block B, each member of the sub-protocol signs B using Pixel with the current block number. The consensus is reached when we see a collection of N committee member signatures $\sigma_1, \ldots, \sigma_N$ on the *same* block B, where N is some fixed threshold. Finally, we will aggregate these N signatures into a single multi-signature Σ, and the pair (B, Σ) constitute a so-called block certificate and the block B is appended to the blockchain.

Registering public keys. Each user who wishes to participate in consensus needs to register a participation signing key. A user first samples a Pixel key pair and generates a corresponding PoP. The user then issues a special transaction (signed under her spending key) registering the new participation key. The transaction includes PoP. PoS verifiers who are selected to run an agreement at round r, check (a) validity of the special transaction, and (b) validity of PoP. If both checks pass, the user's account is updated with the new participation key. From this point, if selected, the user signs on blocks using Pixel.

Vote generation. To generate a vote on a block number t, users first update their keys to correspond to the round number. Subsequently, they sign the block using the correct secret key and propagate the signature to the network.

Propagating and aggregating signatures. Individual committee signatures will be propagated through the network until we see N committee member signatures on the same block B. Note that Pixel supports non-interactive and incremental aggregation: the former means that signatures can be aggregated by any party after broadcast without communicating with the original signers, and the latter means that we can add a new signature to a multi-signature to obtain a new multi-signature. In practice, this means that propagating nodes can perform intermediate aggregation on any number of committee signatures and propagate the result, until the block certificate is formed. Alternatively, nodes can aggregate all signatures just before writing a block to the disk. That is, upon receiving enough certifying votes for a block, a node can aggregate N committee members' signatures into a multi-signature and then write the block and the certificate to the disk. To speed up verification of individual committee member signatures, a node could pre-compute $e(h, y)$ for the y's corresponding to the users with the highest stakes.

Key updates. When using Pixel in block-chains, time corresponds to the block number or sub-steps in consensus protocols. Naively, when associating time with block numbers, this means that all eligible committee members should update their Pixel secret keys for each time a new block is formed and the round number is updated. Assume for simplicity that each committee member signs at most one block (if not, simply append a counter to the block number and use that as the time). If a user is selected to be on the committee at block number t, it should first update its key to sk_t (Pixel supports "fast-forward" key updates from sk_t to $sk_{t'}$ for any $t' > t$), and as soon as it signs a block, updates its key to sk_{t+1} and then propagates the signature. In particular, there is no need for key updates when a user is not selected to be on the committee.

Tweaking the scheme. The blockchain stores Pixel public keys of all eligible committee members, as well as multi-signatures on each block. It is easy to see that we can tweak the Pixel scheme so that public keys live in the group $\mathbb{G}_1$ (which has a more compact description) instead of $\mathbb{G}_2$; this way, we can minimize the size of the blockchain as well as the cost of aggregate verification, which is dominated by the cost of multiplying N public keys for large N. This change does come at a small cost since signing is performed over the slower $\mathbb{G}_2$ instead of $\mathbb{G}_1$. When instantiated with the BLS12-381 pairing-friendly curve, each public key is 48 bytes, and each multi-signature is 48+96=144 bytes independent of N. Moreover, we estimate signing to take less than 3 ms, and signature verification less than 5 ms for $T = 2^{30}$. More details are provided in Section C.

7 Evaluation on Algorand Blockchain

In order to measure the concrete efficiency gains of Pixel, we evaluate it on the Algorand blockchain [59, 60].

Algorand Overview. Algorand is a Pure PoS (PPOS) system, where each token is mapped to a single vote in the consensus without any explicit bonding [59, 60]. Some users may opt-out from participation, in which case their tokens are excluded from the total number of participating tokens (i.e., the denominator in the weight). Each user maintains an *account state* on-chain that specifies her spending key, balance, consensus participation status, participation key, and other auxiliary information. A user wishing to perform a transaction must sign it with her corresponding secret key. Users run a Byzantine consensus algorithm to agree on a block of transactions following the high-level structure we outlined in the previous section. We call a *block certificate* to denote a collection of votes above a certain threshold approving a block. All users in the network validate and store block certificate (and the corresponding transactions) on disk. We refer to a *node* as a computer system running Algorand client software on the user's behalf.

Verifier Vote Structure and Block Certificates. In Algorand, each valid vote for a block proposal includes (a) a proof that the verifier was indeed selected to participate in the consensus at round r, and (b) a signature on the block proposal. In more detail, each vote includes the following fields:

- Sender identifier which is represented by a unique public key registered on-chain (32 bytes).
- Round and sub-step identifiers (8 bytes).
- Block header proposal (32 bytes).
- A seed used as an input to a VRF function for cryptographic sortition (32 bytes).
- VRF credential that proves that the sender was indeed chosen to sign on the block (96 bytes).
- Forward-secure signature authenticating the vote (256 bytes).

Overall, each vote is about 500 bytes (including some additional auxiliary information), half of which is for the forward-secure signature.

Algorand has two voting sub-steps for each round. In the first sub-step, a *supporting* set (of expected size 3000) of verifiers is chosen to vote on a block proposal. In the second sub-step, a *certifying* set (of expected size 1500) of verifiers is chosen to finalize the block proposal. All verifiers' votes propagate in the network during the agreement, but only the *certifying* votes are stored long-term and sufficient to validate a block in the future. Larger *recovering* set (of expected size 10000) is chosen during a network partition for recovery.

Sig. set size	BM-Ed25519	BM-BLS	Pixel
1	256 B	192 Bytes	144 B
1500	375 KB	141 KB	144 B
3000	750 KB	281 KB	144 B
10000	2.4 MB	938 KB	144 B

Table 1: Total size of signature sets using various forward-secure signature schemes for 2^{32} time periods. BM-Ed25519 is instantiated using Algorand's parameters with $10,000$-ary tree of depth 2. BM-BLS is instantiated using the same parameters with public keys in $\mathbb{G}_1$ and signatures in $\mathbb{G}_2$.

Algorand's Existing Solution to Posterior Corruptions. Algorand solves posterior corruptions using forward-secure signatures instantiated with a d-ary certificate tree [9], which we call BM-Ed25519 for convenience. The root public key of an Ed25519 signature scheme is registered on-chain, and keys associated with the leaves (and subsequently used to sign at each round) are stored locally by the potential verifiers. For each block at round r a verifier must (a) produce a valid certificate chain from the root public key to the leaf associated with r, and (b) signature of the vote under the leaf key. Algorand assumes secure erasures and that users delete old keys from their nodes. BM-Ed25519 is instantiated with 10000-ary tree and depth 2 (supporting approximately $2^{26.6}$ time periods). Ed25519-based signatures have public keys of 32 bytes and 64 bytes signatures. Hence, since a valid certificate chain must include the intermediary public keys, the resulting size of each forward-secure signature is $3 \times 64 + 2 \times 32 = 256$ bytes.

7.1 Efficiency Evaluation

Pixel signatures can serve as a replacement of BM-Ed25519 in Algorand following the same design as outlined Section 6.

Setup. Our experiments are performed on a MacBook Pro, 3.5 GHz Intel Core i7 with 16 GB DDR3. We use Alogrand's open-source implentations of Pixel signatures, VRF functions, Ed25519 signing, and verification [1, 2]. For blockchain applications, since the public key must live on-chain, we choose to place Pixel public keys in $\mathbb{G}_1$, obtaining smaller public keys and faster key aggregation during verification. We set the maximum time epoch to $T = 2^{32} - 1$, which is sufficient to rotate a key every second for 136 years.

Figure 3 shows the runtime of individual Pixel algorithms, aggregation, and object sizes for the BLS12-381 curve [7]. Next, we measure quantities that affect all nodes participating in the system: the size of signature sets, bandwidth, and block verification time. In Pixel, the signature set corresponds to a single multi-signature.

	keygen	key update	sign	aggregate ($N = 1500$)	verify agg. ($N = 1500$)	aggregate ($N = 1500$)	verify agg. ($N = 3000$)	$\|pk\|$	$\|\sigma\|$	$\|sk_t\|$
$pk \in \mathbb{G}_1$	1.03 ms	1.8 ms	2.8 ms	7.2 ms	6.7 ms	13.9 ms	8.3 ms	48 B	144 B	43 kB

Figure 3: Performance figures of the Pixel signature scheme algorithms, and the size of public keys, signatures, and secret keys when using a BLS12-381 curve. N denotes the amount of signatures and keys aggregated, respectively. Maximum number of time periods is $T = 2^{32} - 1$.

Storage Savings. In Table 1, we compare the sizes of signature sets that are propagated (for supporting and verifying votes) and stored (for verifying votes) by all participating nodes. We instantiate BM-Ed25519 with Algorand parameters of 10000-ary and depth 2. For BM-BLS we place the public key in $\mathbb{G}_1$ and signatures in $\mathbb{G}_2$. Since BLS supports aggregation of signatures, we can compress all signatures in a certificate chain and the signature of the block into 96 B (note that the public key in the certificate chain cannot be compressed and adds an additional 96 B per signature). Furthermore, we can compress all signatures across votes. Pixel signatures authenticating a block with 1500 signatures are 2667x and 1003x times smaller than signature sets using BM-Ed25519 and BM-BLS, respectively.

In Figure 4, we show long-term blockchain storage improvements using Pixel signatures. We evaluate storage assuming various number of transactions in each block. Each transaction in Algorand is about 232 bytes. We also assume that the entire expected number of certifying verifiers (1500) are selected for each block. Given today's block confirmation time of just under 4.3 seconds per block, Algorand blockchain should produce 10^6 blocks every ≈ 50 days and 10^8 blocks every ≈ 13 years. Pixel signatures improve blockchain size by about 40% and 20% on blocks packed with 1500 and 5000 transactions, respectively. This improvement translates to smaller overall storage requirements and faster catch-up speed for new nodes.

We clarify that the savings we obtain from Pixel are complementary to those of Vault [44], which is another system built on top of Algorand to improve storage and catch-up speed. In particular, Vault can be used in conjunction with Pixel to obtain further storage savings. Vault creates "jumps" between blocks so that the system can confirm block r knowing only block $r - k$ for some parameter k (e.g., $k = 100$). Instead of downloading every block, a catch-up node in Vault only needs to download every kth block. Even using Vault, users would need to download and store about 10^6 blocks for every ≈ 13 years of blockchain operation.

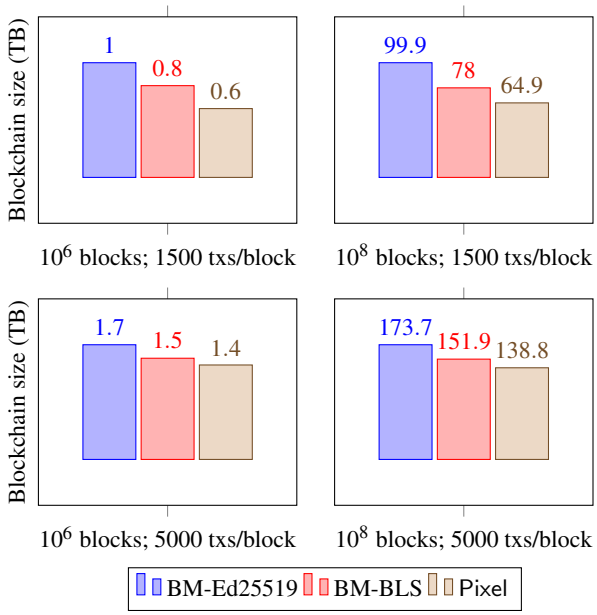

Figure 4: Size of blockchain measured for different total number of blocks. The top two plots assume average of 1500 transactions per block and the bottom plots assume 5000 transactions per block. All plots assume average of 1500 certifying votes per block.

Bandwidth Savings. Algorand uses a relay-based propagation model where users' nodes connect to a network of relays (nodes with more resources). Without aggregation during propagation, Pixel savings for the bandwidth for both relays

Number of connections	BM-Ed25519	Pixel
4	4.4 MB	2.5 MB
10	11 MB	6.2 MB
100	109.9 MB	61.8 MB

Table 2: Total bandwidth to propagate a set of 4500 signatures during consensus to agree on a block of transactions.

Sig. set size	BM-Ed25519	Pixel	Improvement
1	0.18 ms	4.9 ms	27x slower
1500	270 ms	6.7 ms	40x faster
3000	540 ms	8.3 ms	65x faster
10000	1.8 sec	15.6 ms	115x faster

Table 3: Total runtime to verify signature sets authenticating a block. Pixel verification includes the time to aggregate public keys.

and regular nodes come from smaller signatures sizes. Each relay can serve dozens or hundreds of nodes, depending on the resources it makes available. A relay must propagate a block of transactions and the corresponding certificate (with 1500 votes) to each node that it serves. During consensus, however, an additional 3000 *supporting votes* are propagated for every block. Each node connects to 4 randomly chosen relays. Every vote that the node receives from a relay, it propagates to the remaining 3 relays. Duplicate votes are dropped, so each vote propagates once on each connection. In Table 2, we summarize savings for 4500 votes propagated during consensus for each block. From the table, we see that a relay with 10 connections saves about 44% of bandwidth. Bandwidth can be improved even further if Algorand relays were to aggregate multiple votes before propagating them to the users.

Block Verification Time Savings. Since verifying a Pixel multi-signature requires only 3 pairings in addition to multiplying all the public keys in the signature set, they are faster to verify than BM-Ed25519 signatures sets. Table 3 shows that a set of 3000 signatures can be verified about 65x faster. In Figure 5, we measure the overall savings on block verification time. Block verification time is broken into three main intervals: (a) time to verify vote signatures, (b) time to verify vote VRF credentials, and (c) time to verify transactions. In each interval, signature verification dramatically exceeds the time of any additional checks (e.g., check that the transaction amount is higher than the user's balance). Blocks with 1500 and 5000 transactions can be verified 38% and 29% faster, respectively.

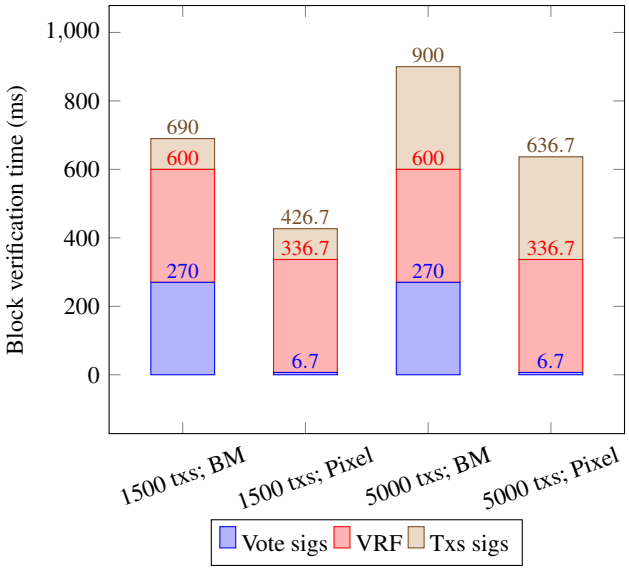

Figure 5: Overall Algorand block verification time using BM-Ed25519 and Pixel signatures. Each block is assumed to contain 1500 certifying votes. The two plots on the left assume 1500 txs/block; whereas the two plots on the right assume 5000 txs/block.

8 Variants and Extensions

Deterministic signatures. In practice, it is helpful to implement deterministic signing and key updates in order to protect against attacks arising from bad randomness. We can achieve using the standard technique [13] of deriving randomness from a random oracle.

More precisely, we assume a random oracle H' that maps to $\mathbb{Z}_q$, and when signing M at time t, we use $r \leftarrow \mathsf{H}'(\texttt{rand-sign}, \tilde{sk}_t, M, t)$. When updating the key from time t to $t+1$, we may need to compute $\widetilde{sk}_{\mathbf{t}\|1}$ and $\widetilde{sk}_{\mathbf{t}\|2}$ from $\widetilde{sk}_{\mathbf{t}}$. The required randomness r can be computed with $\mathsf{H}(\texttt{rand-update}, \widetilde{sk}_{\mathbf{t}})$. Alternatively, if we wish to avoid additional use of a random oracle, we can rely on prior "forward-secure PRG techniques" [43].

Using internal nodes. For ease of exposition, our scheme only assigns time periods to leaf nodes in the tree. Alternatively, one could follow the approach of [24] to use all nodes in the tree, in a pre-order traversal, as time periods, which will improve the efficiency of the key update algorithm. Time periods are then identified by bit strings of length at most ℓ, rather than exactly ℓ bits, and a signature in time period $t = t_1 \ldots t_d$ is a tuple of the form

$$(\sigma_1, \sigma_2) \;=\; \left(h^x \cdot \left(h_0 \prod_{j=1}^{d} h_j^{t_j} \cdot h_{\ell+1}^{\mathsf{H}_q(M)} \right)^r \,,\; g_2^r \right) .$$

Details are left to the reader.

Non-binary trees. One could try to reduce the key size by using b-ary trees instead of binary trees. A larger value of b reduces the depth of the tree, but increases the amount of key material that must be kept at each level of the tree. To support T time periods, one needs a b-ary tree of depth $\ell = \lceil \log_b T \rceil$. A node key at level d, however, can now take up to $b-1$ keys of one element in $\mathbb{G}_2$ and $(\ell + d - 2)$ elements of $\mathbb{G}_1$.

The savings effect is quite limited, however, because the disadvantage of needing more keys per level quickly starts dominating the advantage of having less levels. For practical values of T, the maximum size of the secret key will usually be minimal for $b = 3$.

Parallel key timelines. In some applications, a signer may want to maintain several parallel timelines for different usages of a signing key. For example, in a sharded blockchain, the shards may be running in parallel at different speeds, without strict synchronization between the shards. If a time frame of the forward-secure signature scheme corresponds to the block height of a blockchain, for example, then the signer needs to maintain a different key schedule for the different shards.

A trivial approach is to run a separate instance of Pixel per timeline, and certify each public key with one root signing key. A more efficient approach for our particular scheme is to replace the fixed common parameter h with the output of a hash function $\mathsf{H}_{\mathbb{G}_1}(\texttt{scope}, \textit{scope})$. Meaning, during key generation, the signer generates $sk_{\textit{scope},1} \leftarrow \mathsf{H}_{\mathbb{G}_1}(\texttt{scope}, \textit{scope})^x$ for all relevant scopes *scope* and deletes the master key x. It can then update, sign, and aggregate signatures for each scope separately in the same way as before, but substituting $\mathsf{H}_{\mathbb{G}_1}(\texttt{scope}, \textit{scope})$ for h. Verification of individual signatures and of multi-signatures is also the same as before, substituting $\mathsf{H}_{\mathbb{G}_1}(\texttt{scope}, \textit{scope})$ for h.

Tighter security. The loss in tightness in Equation (3) of $T \cdot q_{\mathrm{H}}$ can be brought down to $T \cdot q_{\mathrm{S}}$ using Coron's technique [29], where q_{S} is the number of signing queries made by the adversary $\mathcal{A}$, by hashing the message into $\mathbb{G}_1$ instead of into $\mathbb{Z}_q$. Namely, a multi-signature would be a tuple $(\Sigma_1, \Sigma_2, \Sigma_3)$ satisfying

$$\mathrm{e}(\Sigma_1, g_2) = \mathrm{e}(h, Y) \cdot \mathrm{e}\big(h_0 \cdot \prod_{j=1}^{\ell} h_j^{t_j}, \Sigma_2\big) \cdot \mathrm{e}\big(\mathsf{H}_{\mathbb{G}_1}(\texttt{msg}, M), \Sigma_3\big) .$$

This scheme has the additional advantage of saving up to ℓ elements of $\mathbb{G}_1$ in secret key size, but signatures are one element of $\mathbb{G}_2$ longer than the base scheme. We leave details to the reader.

Avoiding proofs-of-possession. In situations where proofs-of-possession are not desirable, one could alternatively reuse techniques from [16, 49] to avoid rogue-key attacks. Signers' public keys are simply given by $pk_i = y_i = g_2^{x_i}$, but the aggregate public key is computed as $apk \leftarrow \prod_{i=1}^{n} pk_i^{\mathsf{H}_q(\{pk_1,\ldots,pk_n\},pk_i)}$. Individual signatures $(\sigma_{1,1}, \sigma_{1,2}), \ldots, (\sigma_{n,1}, \sigma_{n,2})$ are aggregated as

$$(\Sigma_1, \Sigma_2) \leftarrow \Big(\prod_{i=1}^{n} \sigma_{i,1}^{\mathsf{H}_q(\{pk_1,\ldots,pk_n\},pk_i)}, \prod_{i=1}^{n} \sigma_{i,2}^{\mathsf{H}_q(\{pk_1,\ldots,pk_n\},pk_i)}\Big) ,$$

so that verification can be performed as usual.

9 Conclusion

In this work, we focus on improving the speed and security of PoS consensus mechanisms via optimizing its core building block – digital signature scheme. We design a new pairing-based forward-secure multi-signature scheme, Pixel. We prove that Pixel is secure in the random oracle model under a variant of Diffie-Hellman inversion problem over bilinear groups. Pixel is efficient as a stand-alone primitive and results in significant performance and size reduction compared to the previous forward-secure signatures applied in settings where multiple users sign the same message (block). For instance, compared to a set of 1500 tree-based forward-secure signatures, a single Pixel signature that can authenticate the entire set is 2667x smaller and can be verified 40x faster. We explained how to integrate Pixel to any PoS blockchains to solve posterior corruptions problem. We also demonstrate that Pixel provides significant efficiency gains when applied to Algorand blockchain. Pixel signatures reduce the size of Algorand blocks with 1500 transactions by $\approx 35\%$ and reduce block verification time by $\approx 38\%$.

Acknowledgments

We would like to thank Zhenfei Zhang for implementing Pixel as well as his help with Section 7. In addition, we thank Jens Groth, Nickolai Zeldovich, our shepherd Ari Juels, and the anonymous reviewers for useful feedback.

References

[1] Algorand's official implementation in go. `https://github.com/algorand/go-algorand`, 2019.

[2] Algorand's official pixel implementation. `https://github.com/algorand/pixel`, 2019.

[3] Ross Anderson. Two remarks on public-key cryptology. Manuscript. Relevant material presented by the author in an invited lecture at the 4th ACM Conference on Computer and Communications Security, CCS 1997, Zurich, Switzerland, April 1–4, 1997, September 2000.

[4] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolic, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: a distributed operating system for permissioned blockchains. In Rui Oliveira, Pascal Felber, and Y. Charlie Hu, editors, *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018*, pages 30:1–30:15. ACM, 2018.

[5] Christian Badertscher, Peter Gaži, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 913–930, New York, NY, USA, 2018. ACM.

[6] Ali Bagherzandi, Jung Hee Cheon, and Stanislaw Jarecki. Multisignatures secure under the discrete logarithm assumption and a generalized forking lemma. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *ACM CCS 2008*, pages 449–458. ACM Press, October 2008.

[7] Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. Constructing elliptic curves with prescribed embedding degrees. In Stelvio Cimato, Clemente Galdi, and Giuseppe Persiano, editors, *SCN 02*, volume 2576 of *LNCS*, pages 257–267. Springer, Heidelberg, September 2003.

[8] Mihir Bellare, Juan A. Garay, and Tal Rabin. Fast batch verification for modular exponentiation and digital signatures. In Kaisa Nyberg, editor, *EUROCRYPT'98*, volume 1403 of *LNCS*, pages 236–250. Springer, Heidelberg, May / June 1998.

[9] Mihir Bellare and Sara K. Miner. A forward-secure digital signature scheme. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 431–448. Springer, Heidelberg, August 1999.

[10] Mihir Bellare and Gregory Neven. Multi-signatures in the plain public-key model and a general forking lemma. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM CCS 2006*, pages 390–399. ACM Press, October / November 2006.

[11] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *ACM CCS 93*, pages 62–73. ACM Press, November 1993.

[12] Iddo Bentov, Rafael Pass, and Elaine Shi. Snow white: Provably secure proofs of stake. Cryptology ePrint Archive, Report 2016/919, 2016. `http://eprint.iacr.org/2016/919`.

[13] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, September 2012.

[14] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In Yvo Desmedt, editor, *PKC 2003*, volume 2567 of *LNCS*, pages 31–46. Springer, Heidelberg, January 2003.

[15] Dan Boneh, Xavier Boyen, and Eu-Jin Goh. Hierarchical identity based encryption with constant size ciphertext. In Ronald Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 440–456. Springer, Heidelberg, May 2005.

[16] Dan Boneh, Manu Drijvers, and Gregory Neven. Compact multi-signatures for smaller blockchains. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part II*, volume 11273 of *LNCS*, pages 435–464. Springer, Heidelberg, December 2018.

[17] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. *Journal of Cryptology*, 17(4):297–319, September 2004.

[18] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A. Kroll, and Edward W. Felten. SoK: Research perspectives and challenges for bitcoin and cryptocurrencies. In *2015 IEEE Symposium on Security and Privacy*, pages 104–121. IEEE Computer Society Press, May 2015.

[19] Xavier Boyen, Hovav Shacham, Emily Shen, and Brent Waters. Forward-secure signatures with untrusted update. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM CCS 2006*, pages 191–200. ACM Press, October / November 2006.

[20] Eric Brier, Jean-Sébastien Coron, Thomas Icart, David Madore, Hugues Randriam, and Mehdi Tibouchi. Efficient indifferentiable hashing into ordinary elliptic curves. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 237–254. Springer, Heidelberg, August 2010.

[21] Vitalik Buterin. Long-range attacks: The serious problem with adaptive proof of work. `https://blog.ethereum.org`, 2014.

[22] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *CoRR*, abs/1710.09437, 2017.

[23] Ran Canetti, Shai Halevi, and Jonathan Katz. A forward-secure public-key encryption scheme. In Eli Biham, editor, *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 255–271. Springer, Heidelberg, May 2003.

[24] Ran Canetti, Shai Halevi, and Jonathan Katz. A forward-secure public-key encryption scheme. *Journal of Cryptology*, 20(3):265–294, July 2007.

[25] Jing Chen, Sergey Gorbunov, Silvio Micali, and Georgios Vlachos. Algorand agreement: Super fast and partition resilient byzantine agreement. Cryptology ePrint Archive, Report 2018/377, 2018.

[26] Jung Hee Cheon. Security analysis of the strong Diffie-Hellman problem. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 1–11. Springer, Heidelberg, May / June 2006.

[27] Sherman S. M. Chow, Lucas Chi Kwong Hui, Siu-Ming Yiu, and K. P. Chow. Secure hierarchical identity based signature and its application. In Javier López, Sihan Qing, and Eiji Okamoto, editors, *ICICS 04*, volume 3269 of *LNCS*, pages 480–494. Springer, Heidelberg, October 2004.

[28] M. Conti, E. Sandeep Kumar, C. Lal, and S. Ruj. A survey on security and privacy issues of bitcoin. *IEEE Communications Surveys Tutorials*, 20(4):3416–3452, Fourthquarter 2018.

[29] Jean-Sébastien Coron. On the exact security of full domain hash. In Mihir Bellare, editor, *CRYPTO 2000*, volume 1880 of *LNCS*, pages 229–235. Springer, Heidelberg, August 2000.

[30] Phil Daian, Rafael Pass, and Elaine Shi. Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake. In *Financial Cryptography and Data Security FC*, 2019.

[31] Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 66–98. Springer, Heidelberg, April / May 2018.

[32] David Derler, Stephan Krenn, Thomas Lorünser, Sebastian Ramacher, Daniel Slamanig, and Christoph Striecks. Revisiting proxy re-encryption: Forward secrecy, improved security, and applications. In Michel Abdalla and Ricardo Dahab, editors, *PKC 2018, Part I*, volume 10769 of *LNCS*, pages 219–250. Springer, Heidelberg, March 2018.

[33] Manu Drijvers, Kasra Edalatnejad, Bryan Ford, Eike Kiltz, Julian Loss, Gregory Neven, and Igors Stepanovs. On the security of two-round multi-signatures. In *2019 IEEE Symposium on Security and Privacy*, pages 1084–1101. IEEE Computer Society Press, May 2019.

[34] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 51–68, New York, NY, USA, 2017. ACM.

[35] Felix Günther, Britta Hale, Tibor Jager, and Sebastian Lauer. 0-RTT key exchange with full forward secrecy. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part III*, volume 10212 of *LNCS*, pages 519–548. Springer, Heidelberg, April / May 2017.

[36] Timo Hanke, Mahnush Movahedi, and Dominic Williams. Dfinity technology overview series, consensus system, 2018.

[37] K. Itakura and K. Nakamura. A public-key cryptosystem suitable for digital multisignatures. Technical report, NEC Research and Development, 1983.

[38] Gene Itkis and Leonid Reyzin. Forward-secure signatures with optimal signing and verifying. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 332–354. Springer, Heidelberg, August 2001.

[39] Thomas Kerber, Aggelos Kiayias, Markulf Kohlweiss, and Vassilis Zikas. Ouroboros crypsinous: Privacy-preserving proof-of-stake. In *IEEE Symposium on Security and Privacy SP*, pages 157–174, 2019.

[40] Thomas Kerber, Markulf Kohlweiss, Aggelos Kiayias, and Vassilis Zikas. Ouroboros crypsinous: Privacy-preserving proof-of-stake. Cryptology ePrint Archive, Report 2018/1132, 2018. `https://eprint.iacr.org/2018/1132`.

[41] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 357–388. Springer, Heidelberg, August 2017.

[42] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016*, pages 279–296. USENIX Association, August 2016.

[43] Hugo Krawczyk. Simple forward-secure signatures from any signature scheme. In Dimitris Gritzalis, Sushil Jajodia, and Pierangela Samarati, editors, *ACM CCS 2000*, pages 108–115. ACM Press, November 2000.

[44] Derek Leung, Adam Suhl, Yossi Gilad, and Nickolai Zeldovich. Vault: Fast bootstrapping for the algorand cryptocurrency. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*, 2019.

[45] Steve Lu, Rafail Ostrovsky, Amit Sahai, Hovav Shacham, and Brent Waters. Sequential aggregate signatures and multisignatures without random oracles. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 465–485. Springer, Heidelberg, May / June 2006.

[46] Changshe Ma, Jian Weng, Yingjiu Li, and Robert H. Deng. Efficient discrete logarithm based multi-signature scheme in the plain public key model. *Des. Codes Cryptography*, 54(2):121–133, 2010.

[47] Di Ma and Gene Tsudik. Forward-secure sequential aggregate authentication. In *2007 IEEE Symposium on Security and Privacy (S&P 2007)*, pages 86–91. IEEE Computer Society, 2007.

[48] Tal Malkin, Daniele Micciancio, and Sara K. Miner. Efficient generic forward-secure signatures with an unbounded number of time periods. In Lars R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 400–417. Springer, Heidelberg, April / May 2002.

[49] Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille. Simple schnorr multi-signatures with applications to bitcoin. *Des. Codes Cryptography*, 2019.

[50] Silvio Micali. ALGORAND: the efficient and democratic ledger. *CoRR*, abs/1607.01341, 2016.

[51] Silvio Micali, Kazuo Ohta, and Leonid Reyzin. Accountable-subgroup multisignatures: Extended abstract. In Michael K. Reiter and Pierangela Samarati, editors, *ACM CCS 2001*, pages 245–254. ACM Press, November 2001.

[52] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system," http://bitcoin.org/bitcoin.pdf, 2008.

[53] Kazuo Ohta and Tatsuaki Okamoto. A digital multisignature scheme based on the Fiat-Shamir scheme. In Hideki Imai, Ronald L. Rivest, and Tsutomu Matsumoto, editors, *ASIACRYPT'91*, volume 739 of *LNCS*, pages 139–148. Springer, Heidelberg, November 1993.

[54] Rafael Pass and Elaine Shi. Thunderella: Blockchains with optimistic instant confirmation. In *EUROCRYPT*, pages 3–33, 2018.

[55] Andrew Poelstra. On stake and consensus. `https://download.wpsoftware.net/bitcoin/pos.pdf`, 2015.

[56] Thomas Ristenpart and Scott Yilek. The power of proofs-of-possession: Securing multiparty signatures against rogue-key attacks. In Moni Naor, editor, *EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 228–245. Springer, Heidelberg, May 2007.

[57] David Schwartz, Noah Youngs, and Arthur Britto. The Ripple protocol consensus algorithm. Ripple Labs Inc White Paper, `https://ripple.com/files/ripple_consensus_whitepaper.pdf`, 2014.

[58] N. R. Sunitha and B. B. Amberker. Forward-secure multi-signatures. In Manish Parashar and Sanjeev K. Aggarwal, editors, *Distributed Computing and Internet Technology, 5th International Conference, ICDCIT 2008*, volume 5375 of *Lecture Notes in Computer Science*. Springer, 2009.

[59] Algorand Team. Algorand blockchain features specification version 1.0. Github, 2019.

[60] Algorand Team. Algorand byzantine fault tolerance protocol specification. Github, 2019.

[61] The Elrond Team. Elrond: A highly scalable public blockchain via adaptive state sharding and secure proof of stake. `https://elrond.com/files/Elrond_Whitepaper_EN.pdf`, 2019.

[62] The ZILLIQA Team. The zilliqa technical whitepaper, 2017. `http://docs.zilliqa.com/whitepaper.pdf`.

[63] Riad S. Wahby and Dan Boneh. Fast and simple constant-time hashing to the BLS12-381 elliptic curve. *IACR TCHES*, 2019(4):154–179, 2019. `https://tches.iacr.org/index.php/TCHES/article/view/8348`.

[64] Jia Yu, Fanyu Kong, Xiangguo Cheng, Rong Hao, Yangkui Chen, Xuliang Li, and Guowen Li. Forward-secure multisignature, threshold signature and blind signature schemes. *Journal of Networks*, 5(6):634–641, 2010.

A Security Proof of Forward-Secure Signatures

Proof. We prove the theorem in two steps. First, we show that the scheme is selectively secure when the message space $\mathcal{M} = \{0,1\}^\kappa$ and H_q is the identity function, meaning, interpreting a κ-bit string as an integer in $\mathbb{Z}_q$.

Step 1: sfu-cma. We show that the above scheme with message space $\mathcal{M} = \{0,1\}^\kappa$ and H_q the identity function is sfu-cma-secure under the ℓ-wBDHI$^*_3$ assumption by describing an algorithm $\mathcal{B}$ that, given a successful sfu-cma forger $\mathcal{A}'$, solves the ℓ-wBDHI$^*_3$ problem. On input $(A_1 = g_1^{\alpha}, A_2 = g_1^{(\alpha^2)}, \ldots, A_\ell = g_1^{(\alpha^\ell)}, B_1 = g_2^{\alpha}, \ldots, \mathcal{B}_\ell = g_2^{(\alpha^\ell)}, C)$, algorithm $\mathcal{B}$ proceeds as follows.

It first runs $\mathcal{A}$ to obtain $(\bar{\mathbf{t}}, \mathbf{t}^*, M^*)$. That is, $\mathcal{A}$ receives $sk_{\bar{\mathbf{t}}}$ and produces a forgery on $\mathbf{t}^*, M^*$. Let $\mathbf{w}^* \in \{0,1,2\}^{\ell-1}$ such that $\mathbf{w}^* = w_1^* \| \ldots \| w_{\ell-1}^* = \mathbf{t}^* \| 0^{\ell-1-|\mathbf{t}^*|}$. It then sets the public

key and public parameters as

$$\begin{aligned} y &\leftarrow B_1 \\ h &\leftarrow g_1^{\gamma} \cdot A_\ell \\ h_0 &\leftarrow g_1^{\gamma_0} \cdot \prod_{i=1}^{\ell-1} A_{\ell-i+1}^{-w_i^*} \cdot A_1^{-M^*} \\ h_i &\leftarrow g_1^{\gamma_i} \cdot A_{\ell-i+1} \qquad \text{for } i = 1, \ldots, \ell \ , \end{aligned}$$

where $\gamma, \gamma_0, \ldots, \gamma_\ell \xleftarrow{\$} \mathbb{Z}_q$.

By setting the parameters as such, $\mathcal{B}$ implicitly sets $x = \alpha$ and $h^x = A_1^{\gamma} \cdot g_1^{(\alpha^{\ell+1})}$. The reduction allows us to achieve two goals:

- extract the value of h^x from a forgery on $\mathbf{t}^*, M^*$ (provided by $\mathcal{A}'$), allowing $\mathcal{B}$ to easily compute its ℓ-wBDHI$_3^*$ solution $\mathrm{e}(g_1, C)^{(\alpha^{\ell+1})}$;
- simulate $\widetilde{sk}_{\mathbf{w}'}$ for all $\mathbf{w}' \in \{0,1,2\}^{\leq \ell-1}$ which are not a prefix of $\mathbf{w}^*$; this would be useful for simulating both the signing and the break-in oracle.

Algorithm $\mathcal{B}$ responds to $\mathcal{A}'$'s oracle queries as follows.

Key update. There is no need for $\mathcal{B}$ to simulate anything beyond keeping track of the current time period $\mathbf{t}$.

Signing. We first describe how to answer a signing query for a message M in time period $\mathbf{t} \neq \mathbf{t}^*$, and then describe the case that $\mathbf{t} = \mathbf{t}^*$ and $M \neq M^*$. Let $\mathbf{w} \in \{0,1,2\}^{\ell-1}$ be such that $\mathbf{w} = \mathbf{t} \| 0^{\ell-1-|\mathbf{t}|}$.

Case 1: $\mathbf{t} \neq \mathbf{t}^*$. It is easy to see that $\mathbf{t} \neq \mathbf{t}^* \Rightarrow \mathbf{w} \neq \mathbf{w}^*$. (This crucially uses the fact that $\mathbf{t}, \mathbf{t}^* \in \{1,2\}^*$.) Then, let $\mathbf{w}' = w_1 \| \cdots \| w_k$ denote the shortest prefix of $\mathbf{w}$ which is not a prefix of $\mathbf{w}^*$. Extending the notation of $\widetilde{sk}_{\mathbf{w}'}$ to $\mathbf{w}' \in \{0,1,2\}^{\leq \ell-1}$, we describe how $\mathcal{B}$ can derive a valid key $\widetilde{sk}_{\mathbf{w}'}$, from which it is straight-forward to derive both $\widetilde{sk}_{\mathbf{w}}$ and a signature for $\mathbf{t}, M$. Recall that $\widetilde{sk}_{\mathbf{w}'}$ has the structure

$$\begin{aligned} &(c, d, e_{k+1}, \ldots, e_\ell) \ = \\ &\left(g_2^r \ , \ h^x \big(h_0 \prod_{i=1}^{k} h_i^{w_i} \big)^r \ , \ h_{k+1}^r \ , \ \ldots \ , \ h_\ell^r \right) \end{aligned}$$

for a uniformly distributed value of r. Focusing on the second component d first, we have that

$$\begin{aligned} d \ &= \ h^x \cdot \left(h_0 \cdot \prod_{i=1}^{k} h_i^{w_i} \right)^r \\ &= \ \left(g_1^{\gamma} A_\ell \right)^{\alpha} \cdot \left(\left(g_1^{\gamma_0} \prod_{i=1}^{\ell-1} A_{\ell-i+1}^{-w_i^*} A_1^{-M^*} \right) \cdot \prod_{i=1}^{k} \left(g_1^{\gamma_i} \mathcal{A}_{\ell-i+1} \right)^{w_i} \right)^r \\ &= \ A_1^{\gamma} g_1^{(\alpha^{\ell+1})} \cdot \left(g_1^{\gamma_0 + \Sigma_{i=1}^{k} \gamma_i w_i} A_{\ell-k+1}^{w_k - w_k^*} \cdot \prod_{i=k+1}^{\ell-1} A_{\ell-i+1}^{-w_i^*} A_1^{-M^*} \right)^r , \end{aligned}$$

where the third equality holds because $w_i = w_i^*$ for $1 \leq i < k$ and $w_k \neq w_k^*$. (Note that in the product notation $\prod_{i=k+1}^{\ell-1}$ above, we let the result of the product simply be the unity element if $k+1 > \ell-1$.) Let us denote the four factors between parentheses in the last equation as F_1, F_2, F_3, and F_4, and denote their product as F. If we let

$$r \leftarrow r' + \frac{\alpha^k}{w_k^* - w_k} \bmod q$$

for a random $r' \xleftarrow{\$} \mathbb{Z}_q$, then we have that

$$d \ = \ A_1^{\gamma} \cdot g_1^{(\alpha^{\ell+1})} \cdot F^{r'} \cdot F^{\frac{\alpha^k}{w_k^* - w_k}} \ .$$

The first and third factors in this product are easy to compute. The second factor would allow $\mathcal{B}$ to compute the solution its ℓ-wBDHI$_3^*$ problem as $\mathrm{e}(g_1^{(\alpha^{\ell+1})}, C)$, so $\mathcal{B}$ cannot simply compute it. The last factor $F^{\frac{\alpha^k}{w_k^* - w_k}}$ can be written as the product of

$$\begin{aligned} F_1^{\frac{\alpha^k}{w_k^* - w_k}} \ &= \ A_k^{\frac{\gamma_0 + \Sigma_{i=1}^{k} \gamma_i w_i}{w_k^* - w_k}} \\ F_2^{\frac{\alpha^k}{w_k^* - w_k}} \ &= \ A_{\ell-k+1}^{-\alpha^k} \ = \ g_1^{-(\alpha^{\ell+1})} \\ F_3^{\frac{\alpha^k}{w_k^* - w_k}} \ &= \ \prod_{i=k+1}^{\ell-1} A_{\ell+k-i+1}^{\frac{-w_i^*}{w_k^* - w_k}} \ = \ \prod_{i=0}^{\ell-k-2} A_{\ell-i}^{\frac{-w_{k+2+i}^*}{w_k^* - w_k}} \\ F_4^{\frac{\alpha^k}{w_k^* - w_k}} \ &= \ A_{k+1}^{\frac{-M^*}{w_k^* - w_k}} \ . \end{aligned}$$

Because $1 \leq k \leq \ell - 1$, it is clear that all but the second of these can be computed from $\mathcal{B}$'s inputs, and that the second cancels out with the factor $g_1^{(\alpha^{\ell+1})}$ in d, so that it can indeed compute d this way. The other components of the key are also efficiently computable as

$$\begin{aligned} c \ &= \ g_2^{r'} \cdot B_k^{\frac{1}{w_k^* - w_k}} \\ e_i \ &= \ h_i^{r'} \cdot A_{\ell+k-i+1} \qquad \text{for } i = k+1, \ldots, \ell \\ &= \ h_{k+i}^{r'} \cdot A_{\ell-i} \qquad \text{for } i = 0, \ldots, \ell-k-1 \ . \end{aligned}$$

From this key $(c, d, e_{k+1}, \ldots, e_\ell)$ for $\mathbf{w}'$, $\mathcal{B}$ can derive a key for $\mathbf{w}$ and compute a signature as in the real signing algorithm.

Case 2: $\mathbf{t} = \mathbf{t}^*, M \neq M^*$. For a signing query with $\mathbf{t} = \mathbf{t}^*$ but $M \neq M^*$, $\mathcal{B}$ proceeds in a similar way, but derives the signature (σ_1, σ_2) directly. Algorithm $\mathcal{B}$ can generate a valid signature using a similar approach as above, but using the fact that $M \neq M^*$ instead of $w_k \neq w_k^*$. Namely, letting

$\mathbf{w} = \mathbf{t}\|0^{\ell-1-|\mathbf{t}|}$, $\mathcal{B}$ computes a signature

$$\begin{aligned}
\sigma_1 &= h^x \cdot \left(h_0 \cdot \prod_{i=1}^{\ell-1} h_i^{w_i} \cdot h_\ell^M \right)^r \\
&= \left(g_1^\gamma A_\ell\right)^\alpha \cdot \left(\left(g_1^{\gamma_0} \cdot \prod_{i=1}^{\ell-1} A_{\ell-i+1}^{-w_i^*} \cdot A_1^{-M^*} \right) \cdot \right. \\
&\quad \left. \prod_{i=1}^{\ell-1} \left(g_1^{\gamma_i} \cdot A_{\ell-i+1} \right)^{w_i} \cdot \left(g_1^{\gamma_\ell} \cdot A_1\right)^M \right)^r \\
&= A_1^\gamma \cdot g_1^{(\alpha^{\ell+1})} \cdot \left(g_1^{\gamma_0 + \sum_{i=1}^{\ell-1} \gamma_i w_i + \gamma_\ell M} \cdot A_1^{M-M^*} \right)^r \\
\sigma_2 &= g_2^r
\end{aligned}$$

by setting

$$r \leftarrow r' + \frac{\alpha^\ell}{M^* - M} \bmod q$$

for $r' \xleftarrow{\$} \mathbb{Z}_q$, so that $\mathcal{B}$ can compute (σ_1, σ_2) from its inputs $A_1, \ldots, A_\ell, B_1, \ldots, B_\ell$ similarly to the case that $\mathbf{t} \neq \mathbf{t}^*$.

Break in. Here, $\mathcal{B}$ needs to simulate $sk_{\bar{\mathbf{t}}}$ where $\mathbf{t}^* \prec \bar{\mathbf{t}}$. This in turn requires simulating $\widetilde{sk}_{\mathbf{w}}$ for all $\mathbf{w} \in \Gamma_{\bar{\mathbf{t}}}$. By the first property of $\Gamma_{\bar{\mathbf{t}}}$ (described in Section 4.2), all of these $\mathbf{w}$ are not prefixes of $\mathbf{t}^*$ and also not prefixes of $\mathbf{w}^*$, and we can therefore simulate $\widetilde{sk}_{\mathbf{w}}$ exactly as before.

Forgery. When $\mathcal{A}'$ outputs a forgery (σ_1^*, σ_2^*) that satisfies the verification equation

$$e(\sigma_1^*, g_2) = e(h, y) \cdot e\big(h_0 \cdot \prod_{j=1}^{|\mathbf{t}^*|} h_j^{t_j^*} \cdot h_\ell^{M^*} \,,\, \sigma_2^*\big) \,,$$

then there exists an $r \in \mathbb{Z}_q$ such that

$$\begin{aligned}
\sigma_1^* &= h^\alpha \cdot \big(h_0 \cdot \prod_{i=1}^{|\mathbf{t}^*|} h_i^{t_i^*} \cdot h_\ell^{M^*}\big)^r \\
\sigma_2^* &= g_2^r \,.
\end{aligned}$$

From the way that $\mathcal{B}$ chose the parameters $h, h_0, \ldots, h_\ell$, one can see that

$$\sigma_1^* = A_1^\gamma \cdot g_1^{(\alpha^{\ell+1})} \cdot (g_1^r)^{\gamma_0 + \sum_{i=1}^{|\mathbf{t}^*|} \gamma_i t_i^* + \gamma_\ell M^*}$$

Note that we do not know g_1^r, so we cannot directly extract $g_1^{(\alpha^{\ell+1})}$ from σ_1^*. Instead, observe that we have

$$\begin{aligned}
e(\sigma_1^*, C_2) &= e(A_1^\gamma, C_2) \cdot e(g_1^{(\alpha^{\ell+1})}, C_2) \\
&\quad \cdot e(C_1, \sigma_2^*)^{\gamma_0 + \sum_{i=1}^{|\mathbf{t}^*|} \gamma_i t_i^* + \gamma_\ell M^*} \,,
\end{aligned}$$

from which $\mathcal{B}$ can easily compute its output $e(g_1^{(\alpha^{\ell+1})}, C_2) = e(g_1, g_2)^{(\gamma \alpha^{\ell+1})}$. It does so whenever $\mathcal{A}'$ is successful, so that

$$\mathbf{Adv}^{\ell\text{-wBDHI}^*_3}_{\mathbb{G}_1 \times \mathbb{G}_2}(\mathcal{B}) \geq \mathbf{Adv}^{\text{sfu-cma}}_{\mathcal{FS}}(\mathcal{A}') \,.$$

Step 2: fu-cma. Full fu-cma security for $\mathcal{M} = \{0,1\}^*$ and with $\mathsf{H}_q : \mathcal{M} \to \{0,1\}^\kappa$ modeled as a random oracle then follows because, given an fu-cma adversary $\mathcal{A}$ in the random-oracle model, one can build a sfu-cma adversary $\mathcal{A}'$ that guesses the time period t^* and the index of $\mathcal{A}$'s random-oracle query for $\mathsf{H}_q(M^*)$, and sets $\bar{t} \leftarrow t^* + 1$. If $\mathcal{A}'$ correctly guesses t^*, then it can use $sk_{\bar{t}}$ to simulate $\mathcal{A}$'s signature, key update, and break-in queries after time $\bar{t}$ until $\mathcal{A}$'s choice of break-in time $\bar{t}'$, at which point it can hand over $sk_{\bar{t}'}$.

If $\mathcal{A}'$ moreover correctly guessed the index of $\mathsf{H}_q(M^*)$, and if $\mathcal{A}$ never made colliding queries $\mathsf{H}_q(M) = \mathsf{H}_q(M')$ for $M \neq M'$, then $\mathcal{A}$'s forgery is also a valid forgery for $\mathcal{A}'$. Note that for $\mathcal{A}$ to be successful, it must hold that $\bar{t}' > t^*$, so it must hold that $\bar{t}' \geq \bar{t}$. The advantage of $\mathcal{A}'$ is given by

$$\mathbf{Adv}^{\text{sfu-cma}}_{\mathcal{FS}}(\mathcal{A}') \geq \frac{1}{T \cdot q_\mathrm{H}} \cdot \mathbf{Adv}^{\text{fu-cma}}_{\mathcal{FS}}(\mathcal{A}) - \frac{q_\mathrm{H}^2}{2^\kappa} \,, \quad (3)$$

where q_H is an upper bound on $\mathcal{A}$'s number of random-oracle queries. Together with Equation (3), we obtain the inequality of the theorem statement. □

B Security Proof of Forward-Secure Multi-signatures

Proof. We show how to construct a forger $\mathcal{A}$ for the multi-signature scheme yields a forger $\mathcal{A}'$ for the single-signer scheme of Section 4.3 such that

$$\mathbf{Adv}^{\text{fu-cma}}_{\mathcal{FS}}(\mathcal{A}') \geq \mathbf{Adv}^{\text{fu-cma}}_{\mathcal{FS}}(\mathcal{A}) \,.$$

The theorem then follows from Theorem 1.

Step 1: simulating $\mathcal{A}$'s view. On input the parameters $(T, h, h_0, \ldots, h_\ell)$ and a public key y for the single-signer scheme, the single-signer forger $\mathcal{A}'$ chooses $r \xleftarrow{\$} \mathbb{Z}_q^*$ and stores $(y, \perp, g_1^r)$ in a list L. It computes $y' \leftarrow y^r$ and runs $\mathcal{A}$ on the same common parameters and target public key $pk = y$ and proof $\pi = y'$. Observe that π is indeed a valid proof for pk since $e(y', g_2) = e(\mathsf{H}_{\mathbb{G}_1}(\mathtt{PoP}, y), y)$.

Algorithm $\mathcal{A}'$ answers all of $\mathcal{A}$'s key update, signing, and break-in oracle queries, as well as random-oracle queries for H_q, by simply relaying queries and responses to and from $\mathcal{A}'$'s own oracles. Queries to the random oracle for $\mathsf{H}_{\mathbb{G}_1}$ are answered as follows.

Random oracle $\mathsf{H}_{\mathbb{G}_1}$. On input $(\mathtt{PoP}, z)$, $\mathcal{A}'$ checks whether there already exists a tuple $(z, \cdot, v) \in L$. If so, it returns v. If not, it chooses $r \xleftarrow{\$} \mathbb{Z}_q^*$, computes $v \leftarrow h^r$, adds a tuple (z, r, v) to L and returns v.y

Step 2: extracting a forgery. When $\mathcal{A}$ outputs its forgery

$$(pk_1^*, \pi_1^*, \ldots, pk_n^*, \pi_n^*), M^*, \mathbf{t}^*, \Sigma^*,$$

algorithm $\mathcal{A}'$ first verifies the proofs $\pi_1^*, \ldots, \pi_n^*$ for public keys $pk_1^*, \ldots, pk_n^*$ and computes the aggregate public key apk^*,

creating additional entries in L if necessary. Let $pk_i^* = y_i = g_2^{x_i}$ and $\pi_i^* = y_i'$. Looking ahead, if pk_i^* passes key verification, then we have $y_i' = (h^{x_i})^{r_i}$ and since we know r_i, we will be able to "extract" $h^{x_i} \in \mathbb{G}_1$.

If all keys are valid, then it holds that $y_i' = \mathsf{H}_{\mathbb{G}_1}(\mathtt{P} \circ \mathtt{P}, y_i)^{x_i}$ for all $i = 1, \ldots, n$. Let $apk^* = Y$ be the aggregate public key. From the aggregate verification equation

$$\mathrm{e}(\Sigma_1^*, g_2) \;=\; \mathrm{e}(h, Y) \cdot \mathrm{e}\big(h_0 \cdot \prod_{j=1}^{|\mathbf{t}^*|} h_j^{t_j^*} \cdot h_\ell^{\mathsf{H}_q(M^*)} \;,\; \Sigma_2^*\big)$$

and the fact that $Y = \prod_{i=1}^n y_i = y \cdot g_2^{\sum_{i=1, y_i \neq y}^n x_i}$, we have that

$$\begin{aligned} \mathrm{e}(\Sigma_1^*, g_2) \;&=\; \mathrm{e}(h, y) \cdot \mathrm{e}(h, g_2)^{\sum_{i=1, y_i \neq y}^n x_i} \cdot \\ &\qquad \mathrm{e}\big(h_0 \cdot \prod_{j=1}^{\ell} h_j^{t_j^*} \cdot h_{\ell+1}^{\mathsf{H}_q(M^*)} \;,\; \Sigma_2^*\big) \\ \Leftrightarrow\; \mathrm{e}(\Sigma_1^* \cdot h^{-\sum_{i=1, y_i \neq y}^n x_i}, g_2) \;&=\; \mathrm{e}(h, y) \cdot \\ &\qquad \mathrm{e}\big(h_0 \cdot \prod_{j=1}^{|\mathbf{t}^*|} h_j^{t_j^*} \cdot h_\ell^{\mathsf{H}_q(M^*)} \;,\; \Sigma_2^*\big)\;. \end{aligned}$$

For all $y_i \neq y$, $\mathcal{A}'$ looks up the tuple (y_i, r_i, v_i) in L. We know that $v_i = h^{r_i}$, and hence that $y_i' = h^{r_i x_i}$. By comparing the last equation above to the verification equation of the single-signer scheme, and by observing that $y_i' = h^{r_i x_i}$, we know that the pair

$$\begin{aligned} \sigma_1^* \;&\leftarrow\; \Sigma_1^* \cdot \prod_{i=1, y_i \neq y}^{n} y_i'^{-1/r_i} \\ \sigma_2^* \;&\leftarrow\; \Sigma_2^* \end{aligned}$$

is a valid forgery for the single-signer scheme, so $\mathcal{A}'$ can output $M^*, \mathbf{t}^*, (\sigma_1^*, \sigma_2^*)$ as its forgery. □

C Efficiency Analysis

We let $T = 2^\ell - 1$ denote the maximum number of time periods.

Computational Efficiency. The main operations are key generation, updating the key, signing, aggregating public keys, and verifying signatures.

- Key generation requires 1 exponentiation in each of $\mathbb{G}_1$ and $\mathbb{G}_2$.
- Key verification requires 2 pairings.
- Key update for an arbitrary number of time steps requires ℓ^2 exponentiations and $2\ell^2$ multiplications in $\mathbb{G}_2$ and ℓ exponentiations in $\mathbb{G}_1$; key updates can of course be entirely precomputed, if necessary. Key updates from $\mathbf{t}$ to $\mathbf{t}+1$ require $\ell - |\mathbf{t}|$ exponentiation in $\mathbb{G}_1$ and 1 exponentiation in $\mathbb{G}_2$ (ignoring multiplications) if $|\mathbf{t}| < \ell - 1$, and no group operations if $|\mathbf{t}| = \ell - 1$. On average, this only requires

  $$1/2 \cdot 0 + 1/4 \cdot 2 + 1/8 \cdot 3 + 1/16 \cdot 4 + \cdots \leq 1.5$$

  exponentiations in $\mathbb{G}_1$ and

  $$1/2 \cdot 0 + 1/4 \cdot 1 + 1/8 \cdot 1 + 1/16 \cdot 1 + \cdots \leq 0.5$$

  exponentiation in $\mathbb{G}_2$. That is, irrespective of the maximum number of time periods T, the average work for updating the key does not exceed 1.5 and 0.5 exponentiations in $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively.
- Signing requires 3 exponentiations and 4ℓ multiplications in $\mathbb{G}_1$ and 1 exponentiation in $\mathbb{G}_2$. By precomputing

  $$\begin{aligned} \sigma_{1,1} \;&\leftarrow\; d \cdot \big(h_0 \cdot \prod_{j=1}^{\ell-1} h_j^{t_j}\big)^{r'} \\ \sigma_{1,2} \;&\leftarrow\; e_\ell \cdot h_\ell^{r'} \\ \sigma_2 \;&\leftarrow\; c \cdot g_2^{r'}\;, \end{aligned}$$

  the signature can be computed as $\sigma_1 \leftarrow \sigma_{1,1} \cdot \sigma_{1,2}^{\mathsf{H}_q(M)}$ once the message M is known, bringing the online computation down to a single exponentiation.
- Aggregating N public keys together costs $N-1$ multiplications in $\mathbb{G}_2$. Here, we ignore the cost of verifying proofs of possession, which should only be performed once per public key.
- Verification of a signature requires 3 pairings (or one 3-multi-pairing) and ℓ multiplications and 1 exponentiation in $\mathbb{G}_1$, plus subgroup membership checks for $\mathbb{G}_1$ and $\mathbb{G}_2$.

Space Efficiency. We are mainly concerned with the size of the public parameters, public keys, secret keys, and signatures.

- The public parameters consist of $\ell + 2$ elements of $\mathbb{G}_1$.
- Every public key is a single element of $\mathbb{G}_2$.
- The size of sk_t is $\ell(\ell-1)/2$ elements in $\mathbb{G}_1$ and ℓ elements in $\mathbb{G}_2$.
- A signature consists of one element in $\mathbb{G}_1$ and one element in $\mathbb{G}_2$.