ALGORAND
The Efficient and Democratic Ledger

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

July 6, 2016

Abstract

Algorand is a truly decentralized, new, and secure way to manage a shared ledger. Unlike prior approaches based on *proof of work*, it requires a negligible amount of computation, and generates a transaction history that does not fork with overwhelmingly high probability. This approach cryptographically selects —in a way that is provably immune from manipulations, unpredictable until the last minute, but ultimately universally clear— a set of verifiers in charge of constructing a block of valid transactions. This approach applies to any way of implementing a shared ledger via a tamper-proof sequence of blocks, including traditional blockchains. This paper also presents more efficient alternatives to blockchains, which may be of independent interest.

Algorand significantly enhances all applications based on a public ledger: payments, smart contracts, stock settlement, etc. But, for concreteness, we shall describe it only as a money platform.

Note: This paper is based on the previous one of Gorbunov and Micali, "Democoin: A Publicly Verifiable and Jointly Serviced Cryptocurrency", https://eprint.iacr.org/2015/521 May 30, 2015.

These 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, US62/653,482 US62/352,195

1 Introduction

Money is becoming increasingly virtual. It has been estimated that about 80% of United States dollars today exist only as ledger entries [2]. Other financial instruments are following suit.

In an ideal world, where 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, it is hard to find such an entity in the real world. Accordingly, decentralized cryptocurrencies such as Bitcoin [7] and "smart contract" systems such as Ethereum have been proposed [5]. At the heart of these systems is a shared *ledger* that reliably records a sequence of transactions, as varied as payments and contracts, in a tamperproof way. The technology of choice to guarantee such tamperproofness is the *blockchain*. Blockchains are in fact behind applications such as cryptocurrencies [7], financial applications [5], and the Internet of Things [8]. Several techniques to manage blockchain-based ledgers have been proposed: *proof of work* [7], *proof of stake* [6], *practical Byzantine fault-tolerance* [21], or some combination.

Currently, however, ledgers can be inefficient to manage. For example, Bitcoin's *proof-of-work* approach requires a vast amount of computation, is wasteful and scales poorly. In addition it *de facto* concentrates power in very few hands.

We thus wish to put forward a new way 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 each one of these selections is provably immune from manipulations, unpredictable until the last minute, but ultimately universally clear.

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

To better appreciate these advantages, let us review the limitations of prior approaches.

2 Prior Problems and Concerns

Prior decentralized payment systems are ingenious, but also problematic. Let us consider the case of Bitcoin, and its many variants. 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. This sequence of blocks is a *chain* because it is structured so as to ensure that any change, even in a single block, percolates in 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*.

Generating a block in Bitcoin, however, requires a great amount of computation. So much so that, even in a *permissioned* setting,[1] Bitcoin must heavily rely on incentives to compensate its users for their very expensive computational efforts.

[1] In a permissioned system, only carefully vetted users are permitted to join. Typically, such vetted users could be relied to act in a prescribed way without any significant rewards. By contrast, in a "permissionless" setting, all users are free to join at any time. Accordingly, here incentives become crucial to encourage a prescribed behavior.

Bitcoin's approach is quickly recalled in Appendix A. Here, we just discuss the assumption it makes, the general problems it suffers from, and some of the personal concerns it has generated. For these limited purposes, it suffices to say that, in Bitcoin, a user may own multiple public keys of a digital signature scheme, that money is associated to public keys, and that a payment is a digital signature transferring some amount of money money from a public key to another.

Honest Majority of Computational Power. Bitcoin assumes that no malicious entity (or a coalition of coordinated malicious entities) E controls the majority of the computational power devoted to block generation. Else, such an E would be able to modify the blockchain, and thus re-write the payment history, as it pleases. In particular, therefore, it could make a payment P, obtains the benefits paid for, and then "erase" any trace of P. (See Appendix A for further details.)

Bitcoin suffers from three main problems. The first two are (interrelated) scalability problems.

Problem 1: Computational Waste Bitcoin's "proof of work" block generation process requires an extraordinary amount of computation. Currently, with just a few hundred thousands public keys in the systems, 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 if should substantially more users join the system. △

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

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

Problem 3: Ambiguity In Bitcoin, the blockchain is not necessarily unique. The end of the chain may *fork*, so that one user may observe that the blockchain is —say— $B_1, \ldots, B_k, B'_{k+1}, B'_{k+2}$ and another may observe that it is $B_1, \ldots, B_k, B''_{k+1}, B''_{k+2}, B''_{k+3}$. Some time later, after a few more blocks have been added, will users agree on blocks B_{k+1} and B_{k+2}. Accordingly, the payee of a payment contained in one of the last blocks cannot be sure that he has actually been paid. The last few blocks may indeed be replaced with others containing different payments. △

Quite separately, two concerns have been raised about Bitcoin. In fairness, however, these concerns are not technological weaknesses, but simply the other side of the coin of the basic properties of Bitcoin: what is an advantage to some, may be a disadvantage to others.

Law-Enforcement Concerns. The (pseudo) anonymity offered by Bitcoin payments may be misused for money laundering and/or the financing of criminal individuals or terrorist organizations.

In principle, traditional banknotes or gold bars, that actually 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 to law enforcement agencies. The situation may be dramatically different for a significantly anonymous digital currency. △

MONETARY-POLICY CONCERNS. The ability to "print money" is one of the very basic powers of a nation state. In principle, therefore, the massive adoption of a convenient and independently floating currency may curtail this power. At this level of adoption, however, Bitcoin is far from being a threat to governmental monetary policies. And, due to its scalability problems discussed above, it may never be. △

3 Preliminaries

3.1 Cryptographic Background

Digital Signatures. Digital signatures are a powerful way to authenticate information, because they do not require any shared 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.

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

Player x uses SK_x to digitally sign messages. For each possible message (binary string) m, x runs algorithm S on inputs m and SK_x in order to produce a string, denoted by SIG_{PK_x} or $SIG_x(m)$ —if x has a single public key— referred to as a digital signature of m relative to PK_x or x's digital signature of m.

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

The properties we require from a digital signature scheme are:

0. *Retrievability:* For all strings m, m is readily computable from $SIG_x(m)$.[2]

1. *Legitimate signatures are always verified:* If $s = SIG_x(m)$, then $V(PK_x, m, s) = YES$; and

2. *Digital signatures are very hard to forge:* Without knowledge of SK_x finding a string s such that $V(PK_x, m, s) = YES$, for a message m never signed by x, requires an astronomical amount of time.

3. *Guaranteed Uniqueness.* For all strings PK' and m, there exists at most one string s such that $V(PK', m, s) = 1$.

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

[2]This is without loss of generality, since a digital signature of m could always be defined so as to include m itself.

Ideal Hashing. We shall rely on an efficiently computable cryptographic hash function, H, mapping 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, and in many implementations of Bitcoin, 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, then finding any such pair of strings is indeed hard. (Trying at random, and relying on the birthday paradox, would require $2^{256/2} = 2^{128}$ trials.)

3.2 The Idealized Payment System

Let us start by describing an abstract system, the *idealized system*, that Algorand tries to mimic.

1. *The Initial Status.* Money is associated to 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 P 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,

$$P = SIG_{PK}(PK, PK', a', I, H(\mathcal{I})),$$

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

   Note that users may join the system whenever they want by generating their own public/secret key pairs. Accordingly, the public key PK' appearing in the payment P above may be a newly generated public key that never "owned" any money before.

3. *The Magic Ledger.* In the ideal system, no payments occur simultaneously, and all payments are valid. As soon as each payment is made, it is magically and immediately appended to a non-tamperable list L, "magically posted on the sky" for everyone to see. This list comprises all payments made so far, together with the time at which they occur:

$$L = (P_1, t_1), (P_2, t_2), \ldots \ .$$

Discussion.

- *More General Payments and Unspent Transaction Output.* More generally, if a public key PK owns an amount a, then a valid payment P 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 P made by PK must transfer such an amount a in its entirety. If PK wishes to transfer only a fraction $a' < a$ of this amount to another key, then it must also transfer the balance, the *unspent transaction output*, to another key, possibly PK itself.

  It is trivial to adopt this approach also in our ideal system and Algorand. But, in order to focus on the novel aspects of Algorand, we prefer to stick our simpler forms of payments.

- *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. (The same is true for the ledger of Bitcoin.)

  In the ideal system, an active user continually stores and updates the latest status information. Else, he would have to reconstruct it from scratch, or from the last time he computed it. In Appendix E, however, 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 P, its public keys and amount are not hidden, but its sensitive information $\mathcal{I}$ is. Indeed, only $H(\mathcal{I})$ appears in P, 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 P. In fact, H is *collision resilient,* and thus it is hard to find a second value $\mathcal{I}'$ such that $H(\mathcal{I}) = H(\mathcal{I}')$.

3.3 Basic Notions and Notations

Users and Keys We identify each public key PK with the user owning it. Accordingly, we can say that PK is honest to mean that its owner is honest, that is, that he follows every instruction prescribed in Algorand. We can also say that PK sends or receives a message. And so on.

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.

Clocks We assume that there is a global clock readable by everyone. (It actually suffices to assume each key X reads its own clock that is at most —say— one second off from a global clock.)

Rounds Algorand is organized in time intervals, called *rounds*. Initially, rounds can be thought as being of fixed length and non overlapping.[3] (One-minute rounds suffice in realistic applications.)

[3] In general, the duration of a round depends on the current number of users and the payment rate. At the payment rate of Bitcoin, one minute should be plenty, in Algorand, to accommodate millions of users. Yet, a round can be

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), we write $Q^{(r)}$ in order to avoid that the symbol r could 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

$$S^{r-1} = \left\{ \left(X, a_X^{(r-1)}, \ldots \right) : X \in PK^r \right\},$$

where $a_X^{(r-1)}$ is the amount of money available to the public key X at the beginning of round r, and "room" is kept for other components as well.

For round 0, PK^0 consists of the set of initial public keys, and S^0 is *the initial status*.[4]

Payments In Algorand a round-r payment P of a key $X \in PK^r$ has the same format and semantics as in the Ideal System, except that it also specifies, as its first item, the round r in which it is actually made. That is,

$$P = SIG_{PK}(X, PK', a, I, H(\mathcal{I})) \ .$$

Payment P is individually *valid* if its amount a is less than or equal to $a_X^{(r)}$. A set of round-r payments of X is collectively valid if the sum of the payments' amounts is less than $a_X^{(r)}$.

Paysets In a round r, a *payset* is the union of collectively valid sets of payments made by some users in round r. A payset $\mathcal{P}$ is *maximal* if no superset of $\mathcal{P}$ is a payset in round r.

Official Paysets For every round r, Algorand publicly selects (in a manner that we shall describe later on) a single (possibly empty) payset, PAY^r, the round's *official payset*.

(Essentially, PAY^r represents the round-r payments that have *"really"* happened.)

Round Operations The goal of a round r is to compute, from PAY^r and S^{r-1}, by means of a proper protocol, the official payset PAY^r, the next status S^r, and the next set of public keys PK^{r+1} (i.e., the union of PK^r and the set of all payee keys that appear for the first time in the payments in PAY^r).

Round-Limited Validity Policy By choice, a round-r payment P is not considered valid in a round $r' \neq r$, and thus cannot appear in $PAY^{r'}$.

REMARK. This policy actually removes the burden of having to check whether an identical payment was ever made before. It also allows for conceptual clarity, without practical inconvenience. As we shall see, even perfectly organized malicious users cannot frequently prevent a valid round-r payment from entering PAY^r. Thus, if P does not appear in PAY^r, then its payer can "reissue" the payment, after marking $r + 1$, at the next round. In any case, Algorand can be adapted so as to accommodate different policies.[5]

completed more quickly if no malicious users act in it. Furthermore, some implementation of Algorand will be more efficient if some overlap between consecutive rounds is allowed (see, in particular, Appendix C.3). Finally, to improve the throughput of Algorand, we may want to run concurrently several rounds (see Subsection 6.1).

[4] As in the ideal system, each S^r is a conceptual quantity deducible from the payment history. In appendix E, we shall directly authenticate status information, and make it easier to compute.

[5] For instance, similarly to Bitcoin, Algorand may allow payments to be processed at any round. Alternatively,

Permissioned and Permissionless Systems A payment system is *permissionless* if any user is allowed to join it at any time, and *permissioned* otherwise.

3.4 The Adversarial Model

Honest and Malicious Users A user is *honest* if he always follows its prescribed instructions, and *malicious* otherwise.

The Adversary Users cannot become malicious, unless previously attacked by the *Adversary.*

The Adversary is an efficient (technically polynomial-time) algorithm, personified for colorfulness, who can choose to attack any user at any round. If the Adversary attacks a user X in round r, then X becomes malicious only in round $r + 1$.

The Adversary subsumes and perfectly coordinates all malicious users. He receives instantaneously all messages they receive in the protocol, and takes all actions on their behalf, including sending all their messages. The Adversary can have the malicious users deviate their prescribed instructions in arbitrary ways.

This powerful adversary does not have, however, unbounded computational power, and cannot successfully forge the digital signature of a honest user, except with negligible probability. Furthermore, his ability to attack honest users is bounded by one of the following two assumptions:

1. Honest Majority of Users: *A given majority (e.g., 75%) of the users are honest.*

2. Honest Majority of Money: *Honest users own a given majority of the money in the system.*

Discussion. Designing a secure system is much easier in a static adversarial model, when users are born malicious or honest, and remain so forever. This is so even when malicious users can perfectly coordinate their actions. The latter, of course, is a rather pessimistic hypothesis. In reality, perfect coordination, particularly among very many individuals is hard. But, since it is hard to be sure about the level of coordination malicious users may enjoy, we'd better be safe than sorry.

Security in a static adversarial model, however, is not too meaningful. In realistic settings, initially honest players may eventually become malicious, but this transformative process is not instantaneous. Should an Adversary be able to simply "point" at any user, at any time, and immediately coerce them to do his bidding, there would be very little room security.

Our model thus takes a middle road. The adversary can target any user he wants, but, if a round takes a minute, then he needs more than one minute to take control over the user.

Finally, note that the Honest Majority of Money assumption is somewhat related to Bitcoin's Honest Majority of Computational Power assumption, in the sense that, since computational power can be bought with money, if malicious users in Bitcoin owned most of the money, then they could gain the majority of computational power.

Algorand can allow a payment marked r to be processed only in rounds $r, \ldots, r + k$, for some fixed, but suitably large, integer k (e.g., $k = 100$). In light of our promise that no valid payment can be delayed too long, no matter what malicious users might do, this alternative policy
(a) essentially guarantees that the payment will go through without its payer having to take any further action, and
(b) does not make it too onerous to check whether a payment was already made in the past.

3.5 The Communication Model

Algorand envisages a separate underlying communication network that reliably handles the necessary message traffic in a timely fashion. We envisage two main ways of transmitting messages.

1. *Direct Messages.* This way simply enables a user X to send a message m to another user Y.
2. *Message Propagation.* As in Bitcoin, this way enables a user to send a message m to all users in a peer-to-peer ("gossip") fashion, via a *message-propagation protocol*. Essentially, every active user X randomly selects a small number of active users as his *neighbors*. During the propagation of a message m, a user X receiving m for the first time forwards it to each of his neighbors, until he receives an acknowledgement that they have received m. The propagation of m terminates when no user receives m for the first time.

We make the following reliability assumptions: there exist constants δ and π, with $\delta < \pi$, such that

(a) If a user X directly sends a message m to a user Y at a time t, then with overwhelming probability Y receives m by time $t + \delta$.

(b) If an honest user X propagates a message m at a time t, then with overwhelming probability all users receive m by time $t + \pi$.

Let us emphasize that, since the communication network is separate, the above delivery bounds hold no matter what the malicious users might do.

In the direct sending of a message m, of course, there is nothing for the malicious users to do. The delivery of m is solely handled by the network.

In the propagation of a message m, instead, all users are involved. Therefore the malicious players may, in particular, decide not to re-transmit m at all. Yet, since the honest users are the majority, and when they receive m for the first time, they re-transmit to sufficiently many randomly selected neighbors, m will quickly reach all users. (Since malicious users may only choose each other as neighbors, no delivery guarantee is given when the propagation of a message is started by a malicious user.)

REMARK Let us stress even more that the above delivery bounds do *not* imply that, when a protocol instructs a user X to propagate a message m at a prescribed time t, by time $t + \pi$, either m is received by all users or by none of them (as it could certainly happen when X is malicious and does not propagate m at all). Indeed, a malicious X may purposely start propagating m at a time $t' \in (t + \tau)$. Thus, it is also possible that, by time $t + \tau$, m is received only by some, but not all, honest users. Indeed malicious players may exploit delayed propagation in order to generate "dissent" among honest players.

4 Organization of the Following Sections

In Section 5 we present a basic version of Algorand based on blockchains. This version is a permissionless payment system or a separate digital currency. It solves the discussed three problems of Bitcoin, but does not address the also discussed law-enforcement and monetary-policy concerns. This basic version already enjoys a very good performance, but relies on some strong assumptions.

In Section 6, still using blockchains, we improve the basic version of Algorand by removing the need for its strong assumptions.(We also discuss some more specialized cryptographic tools that could be advantageously incorporated in Algorand.)

In Section 7 we show how a permissioned (still blockchain-based) implementation of Algorand that avoids the discussed law-enforcement concerns, while still enjoying some degree of privacy. This version of Algorand naturally yields an efficient, secure, and purely distributed form of payment based on a national currency, thereby avoiding also the discussed monetary policy concerns.

In Appendix C.1, after recalling the notion of a *Byzantine agreement* (BA), we present a novel BA protocol, based on digital signatures, that is much more efficient than prior ones. This protocol certainly enhances the performance of Algorand, but may be of independent interest.

In Appendix 6.5 we also introduce a better way to structure blocks in order to construct a tamperproof public ledger. Unlike blockchains, the new structure enables one to verify the correctness of any given block without having to process all subsequent ones —which is wasteful.

In Appendix E we show how to authenticate and handle the system status efficiently and directly, rather than deducing it from the authenticated payment history.

Finally, in Appendix B, we present some additional alternative versions of Algorand.

5 Basic Algorand

The basic version of Algorand is a permissionless payment system, which relies not only on the Honest Majority of Users assumption, but also on the following assumptions and requirement.

5.1 Special (Temporary) Assumptions and Requirements

1. One-Key-Per-User: *Every user has a single public key.*

   This assumption might be reasonable only in permissioned implementations of Algorand. It enables one to perfectly identify a key with its owner, so as to speak meaningfully of honest majorities, whether of keys or users; and to temporarily ignore so called "Sybil attacks", where —in our application— a malicious user may artificially increase the number of keys he owns in an effort to enhance his chances to disrupt the system.

2. Longer Corruption Time: *The Adversary needs multiple rounds to control an honest user.*

   That is, there exists a small constant $k > 1$ (e.g., $k = 40$) such that, if the Adversary attacks a user X in a round r, then X remains honest at least until round $r + k$.

   In addition, the basic version of Algorand relies on the following requirement

3. Continual Participation: *An honest user participates to each round of the protocol.*

The above assumptions and requirements will no longer be needed in the versions of Algorand of Section 6.

5.2 Intuition

Blocks and Proven Blocks Until Subsection 6.5, the block B^r corresponding to a round r specifies: r itself; the set of payments of round r, PAY^r; and the hash of the previous block, $H(B^{r-1})$. Thus, starting from some fixed block B_0, we have a traditional blockchain:

$$B_1 = (1, PAY^1, H(B^0)), \quad B^2 = (2, PAY^2, H(B^1)), \quad B^3 = (3, PAY^3, H(B^2)), \quad \ldots$$

The blockchain of Bitcoin is different, as each of its blocks must satisfy a special property that makes block generation computationally intensive. On the positive side, however, in Bitcoin the blockchain itself constitutes the tamper-proof public ledger. In Algorand, a block B_r need not satisfy any special property, and its authenticity must be vouched by a separate piece of information, which turns B^r into a *proven block*, $\overline{B^r}$. (We could, of course, include this information in the blocks themselves, but find it conceptually cleaner to keep it separate.) In Algorand, therefore, the Magic Ledger is implemented by the sequence of the proven blocks,

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

No Ambiguities Since Bitcoin's blockchain may fork, its blocks are not quite stable and, at some point, a user may find that a recent block no longer belongs to the current blockchain. Algorand, by contrast, guarantees a *unique* blockchain with overwhelming probability. That is, each block B^r contains only valid payments, is universally known, and is never revised. The worst malicious users might do, even if they had the ability and the time to perfectly coordinate their actions, is to slightly delay some payments made by honest users to become effective.

Another possible source of ambiguity in Bitcoin is the following. A user making a payment P, and not seeing P appearing in the next block or two, does not quite know if he is free to use the amount of money he tendered in P or not. Bitcoin allows such a user to issue a cancellation of P, but in principle neither P nor its cancellation may appear in the next few blocks, in which case the user would find himself in a bind. Algorand eliminates this ambiguity too. Recall that each payment P is marked with the round r in which it is made. Even the basic version of Algorand guarantees that, with reasonably high probability, P will appear in PAY^r. If it does not ("courtesy" of the malicious users), then the payer knows that P will never appear in a future block. Thus, he can decide to use the money in a different way or to reissue the payment with the next round number.

In sum, in Algorand all decisions are final, and safely so.

Acceptable Failure Probability To analyze the security of our system, we need to specify the probability, F, with which we are willing to accept that the system fails. 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. In this paper, we set

$$F = 10^{-12} \ .$$

This probability is actually less than one in a trillion, and we believe it to be adequate in our application. Let us emphasize that 10^{-12} is not the probability with which the Adversary can forge our payments —even though the probability of that happening may be acceptable too! Recall that all payments are digitally signed, and thus the probability of forging a payment is way lower than

10^{-12}, and is in fact essentially 0, if the proper digital signatures are used. The catastrophic event that we are willing to tolerate with probability F is that Algorand's blockchain *forks*.

Notice that, with our setting of F and 1-minute long rounds, a fork is expected to occur in Algorand's blockchain as infrequently as once in 1.87 billion years.

If this is not satisfactory, one can clearly modify Algorand, using prior techniques, so as to report and recover from a fork, which we omit doing in this paper. A more cautious user than us may thus wait that a payment P made to him becomes a few blocks deep before relying on P, so as to reduce the probability that P may "disappear" to 10^{-24}, or even lower.

Ideal and Realistic Objectives Assume for a moment that all users in the system are honest, participate to the prescribed protocol, and act in a timely fashion. Then, every round could have length π, that is, the upperbound to message propagation. Indeed, round r could start at time $r \cdot \pi$, end before time $(r+1) \cdot \pi$, and generate the payset PAY^r as follows. At time $r \cdot \pi$, every user starts propagating every new payment he wishes to make. Since all users are honest, the set of all such payments, PAY^r, is a valid payset. Moreover, by time $(r+1) \cdot \pi$, every user receives all such payments and thus locally computes PAY^r.

In this idyllic setting, the following two crucial properties hold:

1. *Perfect Correctness.* Every PAY^r is a valid payset on which all honest users agree.

2. *Inclusiveness 1.* Each PAY^r includes, with probability 1, all round-r payments of honest users.

In a realistic setting, however, malicious users may try hard to violate either property.

To be sure, malicious users cannot fake any round-r payment of an honest user, because they cannot forge his digital signatures. However, they can propagate their own, invalid round-r payments. Yet, these payments will not fool honest users, who check them against the round information in their possession: namely, the current public keys PK^r and the previous status S^{r-1}. Thus, such invalid payments will not enter the official payset PAY^r computed by an honest user.

A more subtle way for a malicious user M to invalidate the correctness property consists of choosing a valid round-r payment P_M and ensure that only some, but not all, honest users include it in their individually computed PAY^r. For instance, M may start propagating P_M just very close to the end of round r, counting on the fact that honest users ignore any alleged round-r payments received after time $(r+1) \cdot \pi$. By starting to propagate P_M deliberately late, M guarantees that it will be received on time by some but not all honest users. Accordingly, PAY^r will contain P_M according to some good users, but not others. Similarly, user M could choose two distinct payments, P_M and P'_M, that are valid *individually*, but not *together* (because the sum of their amounts exceeds the money owned by M), and cause some honest users to include in PAY^r only P_M, others only P'_M, and others yet none of them.

Of course, guaranteeing perfect correctness alone is trivial: all honest users always chose the official payset PAY^r to be empty. But in this case, the system would have inclusiveness 0. Unfortunately, guaranteeing perfect correctness and inclusiveness 1 seems to be hard in the presence of malicious users. Algorand thus adopts a more realistic objective. Informally, letting h, $h > 1/2$, denote the percentage of users that are honest, the goal of the (basic version of) Algorand is

Guaranteeing, with overwhelming probability, perfect correctness and inclusiveness h.

Privileging correctness over inclusiveness seems the right choice. After all, payments not processed in one round can be processed in the next one. But one cannot afford that different honest users hold different opinions about which payments have been made, and thus about how much money each user owns.

Reliance on Verifiers At the highest level, Algorand delegates choosing the official payset PAY^r, among all paysets of round r, to a selected set of *verifiers*, who act with power in such a choice. Algorand also specifies other important aspects, such as how to guarantee that the right information reaches the selected verifiers, the process by which they reach their decisions, and the way in which their decisions are communicated to all users in the system. But verifier selection is the first crucial aspect of Algorand's overall strategy.

The simplest way of selecting a set of verifiers is of course to choose a single outside entity, V, to act as the sole verifier. But of course, such a choice would yield a fully centralized system, with all its drawbacks. A better and still simple solution would be to rely on a fixed set of verifiers, $V_1, \ldots, V_n$. Yet, no matter what mechanism they use to choose and publicize the official paysets, this approach continues to suffer from at least the following two drawbacks:

1. It makes the system vulnerable to cyber attacks. Indeed, a sufficiently resourceful and determined attacker might eventually gain control of a majority of a small set of verifiers.

2. It makes the system subject to internal corruption. It is well-known that "continual power corrupts" and an all-powerful and never-changing committee of verifiers is no exception.

Algorand relies on an ever changing set of verifiers.

Before proceeding, let us highlight the properties that we deem important to satisfy for verifier-selection mechanism. (We wish to acknowledge that other properties can be deemed sufficient, and in fact describe some alternative choices of properties and corresponding mechanisms in Appendix B.)

Main Desiderata in Verifier Selection We believe that a good mechanism should, for every round, efficiently select only *few* verifiers, *at random* from the set of *all users*, and in a way that is both *unpredictable* until the very last moment, and *universally clear*. We call such a mechanism *cryptographic sortition*.

Selecting a small set of round-r verifiers is crucial for the efficiency of the overall system. In particular, it makes it feasible for the verifiers to communicate intensively with each other —e.g., by having each verifier send a separate message to all other verifiers.

Selecting the verifiers at random is also critical. It guarantees that, if the majority of the potential verifiers are honest, then we can expect that the majority of the round-r verifiers are honest too.

Selecting the verifiers from a very large set of potential verifiers may actually suffice, and Algorand can certainly be used in this way. However, having all potential verifiers be users themselves, and actually have the set of all potential verifiers coincide with the set of all users, not only is in keeping with a decentralized and democratic system, but is also very secure. Indeed, it would be essentially impossible for someone to corrupt a majority of all users.

Selecting the verifiers of a round while maintaining their unpredictability up to the last moment is also important. Notice that randomly selecting the verifiers does not suffice to guarantee such unpredictability. If the few verifiers of a given round r were randomly selected but made known

long in advance, then there would be plenty of time for the Adversary to attack and gain control of all of them without violating the Honest Majority of Users assumption. By contrast, selecting the round-r verifiers only at, say, round $r-1$ prevents the Adversary to gain control of them in time to be harmful. One round later, a totally new random set of verifiers will be selected.

Selecting verifiers in a universally clear way is also important. If it is desirable that the round-r verifiers remain unpredictable up to very close to round r, then, since they must collectively decide the official payset PAY^r, it is crucial that their identities become known by at least round r.

Finally, none of these desiderata would matter if the verifier selection process were too slow. Fortunately, in Algorand, this process requires no interaction and is computationally very efficient.

Verifier Selection at a High Level Let R be a sufficiently long string, randomly and independently chosen in round r, so as to become immediately known to all users. Then, in accordance with the above desiderata, we can select a total set of n verifiers, TV^r, from the set of all current users, by means of the following process.

Let $X_1, X_2, \ldots, X_u$ be the lexicographically ordered sequence of all keys in PK^r (which, by the temporary One-Key-Per-User assumption, coincides with the set of all users at the start of round r). Then, $\lceil \log u \rceil$ bits suffice to uniquely identify a key in PK^r. Accordingly, the first $\lceil \log u \rceil$ bits of R identify the first verifier in TV^r. (Should these bits identify a number higher than u, then they are discarded, and the next $\lceil \log u \rceil$ bits are considered instead.) The following $\lceil \log u \rceil$ bits of R identify the second chosen verifier (if different from the one already chosen, else these bits too are discarded). And so on, until all n verifiers of TV^r have been selected.

We refer to this process as the *natural mechanism* (with string R and set PK^r). In symbols,

$$TV^r \xleftarrow{R} PK^r \ .$$

Actually, the natural mechanism not only randomly selects TV^r from PK^r, but also a random ordering ρ^r of TV^r: in symbols,

$$(\rho^r, TV^r) \xleftarrow{R} PK^r \ .$$

But: *Who chooses R?*

Algorand's answer is simple: *no one*. Each string R is algorithmically generated, by applying the cryptographic hash function H to an unambiguously defined input relative to round r. In the simplest instance, this input would be r itself. Namely, we could consider selecting the verifier set TV^r and a random ordering ρ^r of TV^r as follows:

$$(\rho^r, TV^r) \xleftarrow{H(r)} PK^r \ .$$

Indeed, in our model, $H(r)$ is a 256-bit random string.[6] Furthermore, $H(1), \ldots, H(r), \ldots$, are independent random strings.

A problem with running the natural algorithm with string $H(r)$ to select the verifier set TV^r is that the sequence $TV^r, TV^{r+1}, \ldots$, is, yes, randomly selected, but also easy to predict, because it can be computed in its entirety at the start of the system, that is, in round 0.

[6]The natural algorithm may need more than 256 bits to select a set of verifiers, but one could always stretch $H(r)$ into a much longer pseudo-random bit sequence of high quality. In fact, the natural algorithm may take all the blocks of bits it needs from the sequence $H(r) \circ H(r,1) \circ H(r,2) \circ \cdots$, where "$\circ$" denotes the concatenation operator.

To guarantee the unpredictability of future verifiers, we may choose

$$(\rho^r, TV^r) \overset{H(Q^{r-1})}{\longleftarrow} PK^r \ ,$$

where Q^{r-1} is a quantity universally known at round $r-1$, but hard to predict before then.

Assume for a moment that we have already found such a proper quantity Q^{r-1}, and let us start discussing what the verifiers in TV^r should do after being so selected.

Essential Round Structure Inductively, by the end of round $r-1$, all users know the status S^{r-1} and the set of public keys PK^r. They also know the set of round-r verifiers TV^r, since we have promised that TV^r is algorithmically computed from a quantity Q^{r-1} universally known by the end of round $r-1$.

The goal of round r is to establish, with the help of the round-r verifiers and in a way clear to all users, the new official payset PAY^r and the new quantity Q^r. Then, every user computes, on his own, the new status S^r, the new set of verifiers TV^{r+1}, and the new set of public keys PK^{r+1}.[7]

Round r starts by having the users who wish to make payments propagate their payments. Thus, assuming that the round starts conventionally at time 0, by our network assumption this propagation can be completed by time π, after which time every one is instructed to ignore all round-r payments. We say "can be completed" because, as we discussed, malicious users may purposely start propagating their round-r payments very late, in an effort to generate confusion about PAY^r. Accordingly, the only guarantee we have is that, by time π, all users have received the round-r payments made by honest payers. Some round-r payments made by malicious payers, however, may have been received by some, but not all, honest users.

To cope with this possibility, each round-r verifier $X \in TV^r$ compiles his own list of valid round- r payments, PAY^r_X, comprising, for each user Y from which he has received a round-r payment P (by time π), a maximal set of valid round-r payments of Y. Of course, such a maximal set of valid payments coincides with all round-r payments of Y, if user Y is honest. And again, of course, due to the discussed behavior of the malicious users, PAY^r_X and PAY^r_Z may differ if X and Z are different round-r verifiers.

To reconcile their possibly different paysets PAY^r_X into the single official payset PAY^r, the round-r verifiers rely on the following variant of a well-known type of protocol.

Certified Byzantine Agreement Informally, a traditional Byzantine agreement (BA) protocol consists of a communication protocol, in which every party i —out a total of n known parties— has an initial value v_i consisting of an arbitrary string. Calling (as in our case) a player *honest* if he follows all his prescribed instructions, and *malicious* otherwise, a BA protocol guarantees that, no matter what the malicious players might do, upon termination the following two conditions hold:

1. Every honest player X outputs the same value OUT (possibly the special *null value* $\perp$), and
2. If all honest players start with the same initial value v, then $OUT = v$.

[7] Indeed, recall that PK^{r+1} comprises all the keys already in PK^r together with the new public keys that appear for the first time, as payees, in at least one payment in PAY^r. Under the current one-user-one-key assumption, the set $PK^{(r+1)} \setminus PK^r$ (i.e., the set $PK^{(r+1)}$ "minus" the set PK^r) consists of the users who have just joined the system by generating a new public key X (together with its corresponding secret key) and received a payment via X.

We actually rely on a variant of Byzantine agreement, which we term *certified* Byzantine agreement (CBA). Informally, a CBA protocol satisfies the same property 2 defined above, and property 1 augmented as follows:

1′. Every honest player X outputs the same value OUT (possibly the special *null value* $\perp$), together with a matching certificate, $CERT_X$, proving that OUT is indeed the output of all honest players.

Note that the proofs $CERT_X$ and $CERT_Y$ may be different, for different honest players X and Y, but each one of them proves that OUT is the correct unique output of all honest players.

Certified Byzantine agreement is more precisely defined in Appendix C.1, where we also provide two very different CBA protocols: CBA' and $CBA^\star$.

Round Leaders After computing his own set PAY_X^r, each $X \in TV^r$, together with the other round-r verifiers, executes a pre-specified CBA protocol with initial value PAY_X^r. Upon termination, therefore, all honest verifiers will output the same value v, but such a value v may be the null output $\perp$, rather than a set of valid payments! In fact, all honest players may output $\perp$ when they start with initial values that are all different. In our case, therefore, since the malicious players can easily ensure that the initial values PAY_X^r are different, they can easily ensure that the final output is the null value $\perp$, rather than a set of valid round-r payments PAY^r. This being the case, even a few malicious players may force the protocol described so far to have inclusiveness 0.

To avoid this pitfall, we shall rely, for each round r, on the *round leader*, ℓ^r. Ideally, ℓ^r is a randomly selected user. (Since the verifiers are themselves randomly selected, he could be a randomly selected verifier.) Also ideally, all verifiers should know who ℓ^r is. Accordingly, after determining a proper quantity Q^{r-1}, we could first compute

$$(\rho^r, TV^r) \overset{H(Q^{r-1})}{\longleftarrow} PK^r \ ,$$

and then set ℓ^r to be the first verifier in TV^r according to ρ^r. That is, augmenting our notation,

$$(\ell^r, TV^r) \overset{H(Q^{r-1})}{\longleftarrow} PK^r \ .$$

The role of the round leader is the following. After all round-r payers propagate (or more simply send to ℓ^r) their round-r payments, ℓ^r individually computes his own list $PAY_{\ell^r}^r$ of valid payments. If ℓ^r is honest, $PAY_{\ell^r}^r$ includes, for each round-r payee Y, a maximal set of valid round-r payments of Y (and thus all valid payments of Y, if Y is honest). After computing $PAY_{\ell^r}^r$, ℓ^r sends (or propagates) it to all round-r verifiers, which then run the CBA protocol using, as their respective initial values, the values they actually received from ℓ^r.

Accordingly, when ℓ^r is honest, the initial value of each honest verifier X is payset $PAY_{\ell^r}^r$. Thus, upon termination of the CBA protocol, no matter what the malicious verifiers might do, each honest verifier X outputs PAY_{ℓ^r} together with a certificate $CERT_X$ proving that $PAY_{\ell^r}^r$ is the output of all honest verifiers of round r.

At this point all that remains to do is for each honest verifier X to propagate the pair $(PAY_{\ell^r}, CERT_X)$. Within time π, therefore, every user provably learns (possibly with multiple proofs) that the set PAY_{ℓ^r} is the correct unique output of all honest verifiers of round r, and thus adopts it as the set of valid payment of round r. That is, every user sets $PAY^r = PAY_{\ell^r}$.

Thus,

If each round leader is selected at random among all users,
and if the honest users are a fraction h of all users,
then the system has inclusiveness h.

This inclusiveness can actually be increased by relying on multiple round leaders.[8]

Notice that, although in principle there might be multiple propagations of PAY_ℓ^r (i.e., one for each different proof $CERT_X$), all of them can be merged into a single propagation. In fact, it suffices to modify the propagation protocol as follows: informally, a user Y forwards to all his neighbors only the first set-proof pair $(OUT, CERT_X)$ he receives, and ignores all subsequent ones, whether identical to the first pair or with a different second component. Any such pair suffices to prove that OUT is indeed the correct unique output of all honest round-r verifiers.

Difficulties in Leader-and-Verifiers Selection Let us turn our attention to finding a proper quantity Q^{r-1} to use in the natural mechanism for selecting TV^r and ℓ^r. Recall that we also wish Q^{r-1} to be universally known at round $r-1$ and unpredictable before then. Thus, at first glance, we could choose Q^{r-1} to coincide with PAY^{r-1}. After all, it should be hard in —say— round $r-10$ to know what the official payset of round $r-1$ will be. Accordingly, we could select ℓ^r and TV^r via (the trivially modified) natural mechanism as follows:

$$(\ell^r, TV^r) \overset{H(PAY^{r-1})}{\longleftarrow} PK^r \ .$$

However, a little effort shows that, due to the presence of malicious users, this selection mechanism is insecure.[9] A little more effort shows that letting Q^r coincide with PK^r, S^{r-1}, B^{r-1}, $H(B^{r-1})$, or myriads of combinations of these and other quantities manipulatable by malicious players, also yields an insecure system.

Cryptographic Sortition at Last! Let us finally describe a mechanism for selecting leaders and verifiers in a round so as to satisfy our discussed main desiderata. We do so by describing, in three stages, a proper sequence of quantities $Q = Q^0, Q^1, \ldots$ such that, selecting

$$(\ell^r, TV^r) \overset{H(Q^r)}{\longleftarrow} PK^r$$

by (a) first computing $(\rho^r, TV^r) \overset{H(Q^r)}{\longleftarrow} PK^r$ and then

(b) choosing the ℓ^r to be the first verifier in TV^r according to ρ^r,

[8] For instance, Algorand may use three round leaders, ℓ_1^r, ℓ_2^r and ℓ_3^r. Each leader ℓ_i^r sends to the verifiers in TV^r his own payset, $PAY_{\ell_i^r}^r$. Denote by $PAY_{\ell_i^r}^r[X]$ the first payset that a verifier X actually receives from leader ℓ_i, where $PAY_{\ell_i^r}^r[X] = \emptyset$ if X receives no payset form ℓ_i. Then, in the following CBA protocol, each verifier X sets his initial value to be PAY_X^r, if there exists two different leaders ℓ_i and ℓ_j such that $PAY_{\ell_i^r}^r[X] = PAY_{\ell_j^r}^r[X] = PAY_X^r$, and $\emptyset$ otherwise. Note that the leaders ℓ_i^r and ℓ_j^r are both honest, then $PAY_{\ell_i^r} = PAY_{\ell_j^r}$, $PAY_{\ell_i^r}$ and $PAY_{\ell_j^r}$ contain all round-r payments of honest users, and $PAY_{\ell_i^r}[X] = PAY_{\ell_j^r}[X]$ for all verifiers X. Accordingly, upon termination of the CBA protocol, the official payset PAY^r will include all round-r payments of honest users, if the majority of the leaders is honest, an event whose probability is greater than h.

[9] Assume that the leader of round $r-1$, ℓ^{r-1} is malicious. Accordingly, he could select PAY^{r-1} in any way he wants. In particular, among all payments he sees propagated in round $r-1$ (or can generate himself), ℓ^{r-1} selects one, P', such that, computing $(\ell^r, TV^r) \overset{H(\{P'\})}{\longleftarrow} PK^r$, ℓ^r is also a malicious user. If the malicious users are —say— 10% of all users, then after a mere 10 trials such a payment P' will be found. Thus, by digitally signing $\{P'\}$ and sending it to the verifiers of round $r-1$, ℓ^{r-1} ensures that $PAY^{r-1} = \{P'\}$, and thus that the leader of the next round, ℓ^r, is also malicious. If all malicious leaders operate this way, the inclusiveness of the system would be 0.

we can be sure that, no matter what the malicious players might do,

(a') TV^r is a randomly selected set of n users and
(b') ℓ^r is a randomly selected verifier in TV^r.[10]

STAGE 1. Each Q^r is inductively constructed, via the previous leader ℓ^{r-1}, as follows.

Let Q_0 be an initially chosen, random, 256-bit string. Then, define the following chain

$$\begin{aligned} Q_1 &= H\left(SIG_{\ell^1}\left(Q^0\right), 0\right). \\ Q_2 &= H\left(SIG_{\ell^2}\left(Q^1\right), 1\right). \\ Q_3 &= H\left(SIG_{\ell^3}\left(Q^2\right), 2\right). \\ &\text{Etc.} \end{aligned}$$

That is, the quantity Q^{r-1} is used by the natural mechanism to determine TV^r and ℓ^r, and then the signature of ℓ^r of Q^{r-1}, hashed together with the sequence of the previous quantities, determines the new quantity Q^r.

Note that, due to the guaranteed uniqueness property of the underlying digital signature scheme, the signature $SIG_{\ell^r}\left(Q^{r-1}\right)$ is uniquely determined by Q^{r-1}. Thus, even if the leader ℓ^r were malicious, he could not "shop around", among a set of multiple valid signatures he may have for Q^{r-1}, for one that lets him *de facto* choose the quantity Q^r in a way that is convenient to him.

Also note that, to generate each quantity Q^r, the input provided to the random oracle H has never been provided to H to generate another quantity Q^j. In fact, each input provided to the random oracle H is a pair, whose second entry is an ever increasing counter. Thus, Q^r is a randomly and independently chosen 256-bit string no matter how a malicious ℓ^r might choose his public key.[11]

Finally note that the randomness of each Q^{r-1}, coupled with the fact the natural mechanism, run with string Q^{r-1}, guarantees that ℓ^r is randomly chosen, implies that, under the Majority of Honest Users Assumption,

$$\ell^r \text{ is an honest user with probability at least } 1/2.$$

In turn, since an honest leader ℓ^r will not divulge his signature $SIG_{\ell^r}(Q^{r-1})$ before round r, the above guarantee implies that the malicious users can predict Q^r in round $r-1$ essentially with probability at most $1/2$.

Extending this reasoning, one can see that the malicious players cannot, at a round r', predict the quantity Q^r of a future round r with probability substantially greater than $2^{-(r-r')}$. In fact, for their prediction to be correct, they must either correctly predict a random 256-bit string (which will happen with absolutely negligible probability), or be so lucky that all $r-r'$ leaders $\ell^{r'+1}, \ldots, \ell^r$ are malicious (which will happen with probability $2^{-(r-r')}$), so that they can compute and share at round r' their digital signatures $SIG_{\ell^{r'+1}}(Q^{r'}), \ldots, SIG_{\ell^r}(Q^{r-1})$.

[10] Note that randomly selecting TV^r from PK^r and then letting the leader ℓ be the lexicographically smallest verifier in TV^r does not work. This is so because malicious users could always choose their keys to be lexicographically very very small, so that, no matter what the actually chosen verifier set TV^r may be, the leader will always be a malicious user. Since a malicious leader could always choose the set of payments PAY^r to be empty, the inclusiveness of the resulting system would be essentially 0.

[11] Without this second, ever-increasing entry, this property might not hold, or might need a more careful proof. For instance, a malicious ℓ^r might not choose his public key by honestly running the key generator G, but in a way guaranteeing him the following strange property: $SIG_{\ell^r}(x) = SIG_{\ell^r}(y)$ for all strings x and y. If he succeeded, then $Q^r = Q^{r+1} = \cdots$, and thus $\ell^{r+1} = \ell^{r+2} = \cdots$, opening the possibility that a malicious user be the leader for all future rounds. Note that this strange property is not ruled out by the guaranteed uniqueness property of the underlying digital signature scheme. But it is ruled out by the retrievability property. Of course, there may be further, equally dangerous properties to guard against.

Thus, at each round r, the future quantity Q^{r+40}, and thus the verifier set TV^{r+40}, can be predicted exactly with probability at most 2^{-40}; and with complementary probability, the only knowledge about TV^{r+40} available to the malicious players is that it consists of n randomly selected users. Notice that the probability 2^{-40} is actually lower than 10^{-12}, our acceptable failure probability F. Thus, for our basic version of Algorand to be secure, it is important to assume that the temporary assumption 2 mentioned at the start of Section 5, Longer Corruption Time, actually holds with $k = 40$. Else, the Adversary may, with probability less than F, correctly learn the quantity Q^r at round $r - 39$, immediately compute the verifier set TV^r, immediately attack all of its n members, gain control of all of them at the start of round r, and thus instruct them to act so as to force a fork in Algorand's blockchain.

The Longer Corruption Time assumption with $k = 40$ may be reasonable. After all, with 1-minute long rounds, 40 minutes are hardly enough to totally corrupt $n/2$ honest users. (Indeed, since TV^r has n members and it is randomly selected, at least half of them are expected to be honest.) Nonetheless, as already mentioned, we shall reduce k to 1 in the version of Algorand of Subsection 6.2.

Stage 2. A problem, however, exists. Namely, a malicious leader ℓ^r may refuse to produce the signature $SIG_{\ell^r}\left(Q^{r-1}\right)$ required to compute Q^r. In this case, to prevent that the generation of the sequence of quantities Q gets stuck, the string $SIG_{\ell^r}\left(Q^{r-1}\right)$ is replaced by —say— the value $Q^0 + r$. That is, Q is so defined.

$$\begin{array}{lll} Q_1 = H\left(SIG_{\ell^1}\left(Q^0\right), 0\right) & \text{if } \ell^1 \text{ reveals } SIG_{\ell^1}\left(Q^0\right). & \text{Else, } \quad Q_1 = H\left(Q^0 + 1, 0\right) \\ Q_2 = H\left(SIG_{\ell^2}\left(Q^1\right), 1\right) & \text{if } \ell^2 \text{ reveals } SIG_{\ell^1}\left(Q^1\right). & \text{Else, } \quad Q^2 = H\left(Q^0 + 2, 1\right) \\ Q_3 = H\left(SIG_{\ell^3}\left(Q^2\right), 2\right) & \text{if } \ell^3 \text{ reveals } SIG_{\ell^3}\left(Q^2\right). & \text{Else, } \quad Q_3 = H\left(Q^0 + 3, 2\right) \end{array}$$

Etc.

Let us quickly analyze this new sequence Q. A malicious leader ℓ^r has certainly the opportunity (ignoring all financial disincentives discussed in Section 5.5) to cause the official payset PAY^r to be empty. In addition, by revealing the required signature $SIG_{\ell^r}\left(Q^{r-1}\right)$ or allowing it to be replaced by the value $Q^0 + r$, he has essentially two shots at selecting the next verifier set TV^{r+1}, its ordering ρ^{r+1}, and thus the next leader ℓ^{r+1}. Thus, he has a second chance of forcing ℓ^{r+1} to be a malicious player. Under both chances available to him, however, TV^{r+1}, ρ^{r+1}, ℓ^{r+1} will be randomly selected. Thus, if in both of these "independent trials" the new leader is honest, then there is nothing that ℓ^r could do to prevent ℓ^{r+1} from being honest, and thus for Q^{r+1} to be unpredictable in round r. We conclude that, when the percentage of honest users is h, then the new leader will be honest with probability at least h^2, when the current leader is malicious, and with probability exactly h, when the current leader is honest. Since an honest leader ℓ^r ensures that PAY^r contains all the round-r payments made by honest users, we conclude that

the current selection mechanism has inclusiveness at least h^2.

Thus, if $h > 1/2$ or $h > 3/4$, then the corresponding inclusiveness is at least $1/4$; and if $h > 3/4$, then the corresponding inclusiveness is at least $9/16$.

Let us now clarify how exactly the round-r leader ℓ^r "reveals" $SIG_{\ell^r}\left(Q^{r-1}\right)$. Of course, propagating $SIG_{\ell^r}\left(Q^{r-1}\right)$ does not work. (Indeed, once again, a malicious ℓ^r may start propagating it so late that only some of the honest users will see it. Accordingly, different honest users will hold different views about the quantity Q^{r+1}, and thus about the next verifier set, leader, etc.)

To avoid that a malicious leader ℓ^r may "tamper" with revealing $SIG_{\ell^r}\left(Q^{r-1}\right)$, we again rely on certified Byzantine agreement. For the sake of efficiency, we choose to leverage the same execution of the CBA protocol, used to determine PAY^r, to determine also $SIG_{\ell^r}\left(Q^{r-1}\right)$. Specifically, after the verifier set TV^r and the leader ℓ^r have been selected, and after he has already compiled his own set of valid round-r payments $PAY^r_{\ell^r}$, ℓ^r does not just send (or propagate) to the round-r verifiers the single item $PAY^r_{\ell^r}$, but the pair

$$\left(PAY^r_{\ell^r}, SIG_{\ell^r}\left(Q^{r-1}\right)\right) \ .$$

The round-r verifiers then run the CBA protocol, each verifier using as his initial value the pair he actually received from ℓ^r. Upon termination, an honest verifier $X \in TV^r$ propagates the computed result OUT and its corresponding certificate $CERT_X$, where OUT is the same for all honest verifiers. Thus, all users will have received OUT in certified form. If OUT is a pair whose second component is $SIG_{\ell^r}(Q^{r-1})$, then all users set

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

Else, all users set

$$Q^r \ = \ H\left(Q^0 + r, r-1\right) \ .$$

In particular, therefore, $Q^r \ = \ H\left(Q^0 + r, r-1\right)$ when $OUT = \perp$. The first component of OUT is used as previously discussed in order to determine PAY^r.

Stage 3. However, one more stage is needed due to one last source of insecurity. Namely, even using the last defined quantity Q^{r-1}, and computing

$$(\ell^r, TV^r) \overset{H\left(Q^{r-1}\right)}{\longleftarrow} PK^r \ ,$$

the malicious players can ensure that the inclusiveness of the system is 0. This is so because, although our lastly chosen quantities Q^{r-1} prevent the malicious players from (significantly) manipulating the string $H(Q^r)$ used by the natural mechanism to choose ℓ^r and TV^r from PK^r, the malicious players can manipulate the set PK^r (better said, they can choose the new public keys that will be added to PK^{r-1} to yield PK^r) so as to ensure that a malicious leader ℓ^r will be selected.[12]

To prevent this last manipulation, we randomly choose the leader and the verifiers of round r not from the most recent set of public keys, PK^r, but from that of a few rounds before:

$$(\ell^r, TV^r) \overset{H\left(Q^{r-1}\right)}{\longleftarrow} PK^{r-k} \ ,$$

where k is a sufficiently large integer —e.g., $k = 100$. Let us emphasize that selecting ℓ^r and TV^r among the users of round $r-k$ does not mean that TV^r is predictable at round $r-k$.

[12] Assume that the leader of round $r-1$, ℓ^{r-1}, is malicious. Then, he may honestly sign Q^{r-1} and reveal $SIG\ell^{r-1}(Q^{r-1})$, but tamper with PAY^r and the set of public keys PK^r as follows. Recall that PK^r consists of the union of PK^{r-1}, the set of keys at the start of round $r-1$, and the set of new public keys, that is, those that appear for the first time as payees in PAY^{r-1}. Leader ℓ^{r-1} keeps on choosing a set of valid round-$(r-1)$ payments P', from malicious payers to newly generated malicious keys, so as to compute a set of new keys $\mathcal{PK}'$, and then computing $(\ell^r, TV^r) \overset{H(Q^{r-1})}{\longleftarrow} PK^r \cup \mathcal{PK}'$ until ℓ^{r-1} is malicious. Once he finds such a set of payments $\mathcal{P}'$, he sends it to all verifiers in TV^{r-1} as they were the only payments he saw in round $r-1$. (If he wants, he can actually have the malicious players properly propagate $\mathcal{PK}'$, so as to legitimize his actions.) By so doing, therefore, ℓ^{r-1} ensures that $PAY^{r-1} = \mathcal{P}'$ and that the next leader is malicious too. If all malicious leaders act this way, then the inclusiveness of the system would be 0.

Quick Summary In sum, the high-level organization of a round r is intuitively as follows: (a) the users propagate their round-r payments; (b) the round-r verifiers and the round leader are selected, based on the previous quantity Q^{r-1}, in a way that ensures that the majority of the verifiers are honest and that the round leader is honest with good probability; (c) the round leader sends to all round-r verifiers a pair consisting of a list of valid round-r payments chosen and authenticated by him, and his own digital signature $SIG_\ell\left(Q^{r-1}\right)$; (d) the verifiers reach certified Byzantine agreement on the received pair, so as to output a certified common value v, which coincides with the original pair sent by ℓ, if he was honest; (e) all honest verifiers propagate (indeed, via a single, coalesced propagation) the certified value v; (f) all users compute from the certified v both the set of payments PAY^r and the quantity Q^r.

Since the certified value v is uniquely determined, so is PAY^r. Since PAY^r is a maximal set of valid round-r payments whenever ℓ^r is honest; and since ℓ^r is honest with good probability, the system has essentially perfect correctness and good inclusiveness. Let us now see a round in detail in the basic version of Algorand.

5.3 Precise Description

Basic Quantities. We shall use the following basic quantities:

- h, the percentage of honest users.

  (Quantity h is empirically determined, but assumed to be high: e.g., h =3/4, 2/3, or 60%.)

- F, the acceptable failure probability, set to 10^{-12}, as already mentioned.

- n and integer k, respectively the number of verifiers in a round and a security parameter.

  (These are "dependent" parameters. Indeed, they are chosen so that the probability that something goes wrong in a round is smaller than F in a system whose honest percentage is h.)

Protocol Round(r)

Initial Comment. After ending Round($r-1$), a user starts executing Round(r) already knowing PK^r, S^r, PAY^{r-1}, Q^{r-1}, and the block B^{r-1}.

Communication Step 1. *Each user* $X \in PK^r$

- *Computes* $(\rho^r, TV^r) \overset{H(Q^r)}{\longleftarrow} PK^{r-k}$.
- *Computes* ℓ^r*, the first verifier in* TV^r *according to* ρ^r.
- *Propagates his own set of valid round-r payments, if any.*[13]

Communication Step 2. *The leader* ℓ^r

- *Computes a maximal payset* $PAY^r_{\ell^r}$ *from the round-r payments he receives by time* π.

[13] To minimize message traffic, during the propagation of a message m supposed to have a given form, a user should not forward m to his neighbors, if m does not have the required form. Thus a user should not forward an invalid round-r payment.

- *Propagates* $SIG_{\ell^r}\left(PAY^r_{\ell^r}, SIG_{\ell^r}(Q^{r-1})\right)$ *to all verifiers in* TV^r_{PK}.[14]

COMMUNICATION STEP 3. *Each verifier* $X \in TV^r$

- *Sets* $v_i = \sigma_i$, *if* σ_i *is the first string he receives of the form* $SIG_{\ell^r}\left(PAY, SIG_{\ell^r}(Q^{r-1})\right)$, *where* PAY *is a round-*r *payset.*
  Else, sets $v_i = \perp$.
- *Executes the CBA protocol with initial input* v_i, *so as to output a value* OUT^r *and a matching certificate* $CERT^r_X$.
- *Propagates the "verdict"* $(OUT^r, CERT^r_X)$.

FINAL COMPUTATION STEP. *Each user* U, *upon receiving the first syntactically correct verdict* $(OUT^r, CERT^r_X)$ *in the previous Step 3, acts as follows.*

- *If* $OUT^r = \perp$,
  then U *sets* $PAY^r = \emptyset$ *and* $Q^r = H\left(Q^0 + r, r-1\right)$.
  Else, letting $OUT^r = SIG_{\ell^r}\left(PAY, SIG_{\ell^r}\left(Q^{r-1}\right)\right)$,
  U *sets* $PAY^r = PAY$ *and* $Q^r = H\left(SIG_{\ell^r}(Q^{r-1}), r-1\right)$.
- U *sets the new block to be* $B^r = \left(r, PAY^r, H\left(B^{r-1}\right)\right)$;
- U *sets the new proven block to be* $\overline{B^r} = (B^r, CERT^r_X)$, *and considers round* r *completed.*

FINAL COMMENT. While PAY^r and the block B^r are the same for every user, the certificate $CERT^r$ (and thus the proven block $\overline{B^r}$) may be different for different users,[15] yet these certificates (and proven blocks) are functionally equivalent. That is, every user presented with any proven block $\overline{B^r}$ can rest assured that B^r is the only correct block of round r.

5.4 Basic Performance Analysis

In Algorand, only the time taken by communication matters: the time taken by internal computation is negligible (quite differently from protocols based on proof-of-work) and will be ignored.

We distinguish two measures of performance: *latency*, the time taken to execute a round, and *throughput*, the time between the determination of two consecutive official paysets.

Latency Analysis The backbone of ROUND(r), that is, ROUND(r) except for the CBA subroutine, consists of 3 communication steps. (Step 4 involves only internal computation.)

Step 1 essentially consists of the propagation of the individual payments made in the round.

Step 2 essentially consists of the propagation of the official payset PAY^r to just TV^r.

Step 3 essentially consists of the "consolidated" propagation to all users of $(OUT^r, CERT^r)$, where OUT^r is either PAY^r or $\perp$, and $CERT^r$ consists, as we shall see, of a set of verifier signatures.

[14] Propagation of a message m to a subset S of the users can be implemented by propagating m to all users. In principle, however, one can design a more efficient protocol for propagating m to just S.

[15] This is so because the certificates $CERT_X$ produced by different verifiers X in our certified Byzantine agreement protocol CBA may not be the same.

Accordingly, *in absolute terms,* the latency of the backbone of ROUND(r) is quite reasonable. In fact, the time needed for propagation is logarithmic in the number of the relevant users, which is at most a few billions.

Moreover, this latency is quite reasonable also in *relative terms.* In a sense, a fully decentralized payment system must always include the propagation to all users of `(i)` the payments in some time interval and `(ii)` the publicly recognized set payments in that time interval. To this basic communication, the backbone of *Round(r)* only adds, in Step 2, the propagation of a single payset to few verifiers.

Let us now consider the latency added by the single call to the chosen CBA protocol. This additional latency clearly depends on the number of the verifiers, n, the percentage of honest users, h, the chosen failure probability, F, and the chosen CBA protocol. Let us discuss this latency for two, and dramatically different, CBA protocols.

- *CBA′.*

  CBA′ is essentially the Byzantine agreement protocol of Dolev and Strong [17], a deterministic protocol in which all messages are digitally signed.

  To turn their BA protocol into a CBA protocol, it suffices to (1) include, in each message m, the number of the communication stage and the execution in which m has been sent; and (2) have each honest player X output not only the traditional value OUT, but also a matching certificate $CERT_X$ consisting of all the digital signatures received by X throughout the execution.

  The latency of the so modified protocol is as follows.

  CBA′ requires the number of malicious players to be less that $n/2$. When there are t malicious players, $t < n/2$, it requires $t + 1$ sequential stages of n-to-n communication.[16]

- *CBA⋆.*

  This is a new CBA protocol, presented in Appendix C.3. Its latency is as follows.

  CBA⋆ requires the number of honest players to be greater than $2n/3$; but only 3 sequential stages of n-to-n communication, if the round leader is honest, and 6 (expected) such stages otherwise. Moreover, except for the first two stages, every player sends a single bit to each other player.

  (Protocol *CBA⋆* is conceptually simple. It is an efficient adaptation of the basic BA protocol of Feldman and Micali [22], which indeed required a 2/3 honest majority, an expected 18 waves of n-to-n communication, and no cryptographic authentication, but was very complex. Other complex BA protocols, with an expected constant number of n-to-n communication waves, relying on cryptographic authentication, but requiring only a simple honest majority, have been discovered by Katz and Koo [23]. Also their BA protocols can be adapted so as to work efficiently within Algorand and reduce the number n of verifiers needed. But adapting the protocol of [22] is simpler and suffices for this version of this paper.

The different honest majorities required by these CBA protocols call for different choices of n. For example, if the percentage of honest users is 75% and we wish that each verifier set has the required majority with probability at least $1 - 10^{12}$, then n must be about 50 for Classical CBS and about 500 for *CBA⋆*. Accordingly, CBA′ requires 26 50-to-50 exchanges, and *CBA⋆* requires three (if the leader is honest, and expected 6 otherwise) waves 500-to-500 message exchanges.

[16] In such a stage, each verifier i sends a messages m_{ij} to each verifier j.

Since the latency of a protocol is also affected by the length of the messages exchanged, let us thus emphasize that, in both CBA' and $CBA^\star$, only the initial messages may be long, since they consist of paysets. All others essentially consist of hashed values and verifier signatures, and are quite short.

Which of these CBA protocols has the best latency performance is not clear (and may ultimately depend on the very specifics of the underlying communication network). Initial simulations indicate that, given the size of the envisaged messages, the envisaged number of verifiers, and the envisaged length of a round, as it is often "the number of sequential waves matters more than bandwidth" and that using $CBA^\star$ provides the better performance [25].

Throughput Analysis In the basic version of *Algorand*, latency equals throughput. Indeed, the users start working on PAY^{r+1} only after PAY^r has been determined, with just a minor overlap at the round transition. Such minor overlap occurs because some users may see slightly earlier than others that a round has terminated (either because the final propagated message reaches them earlier, or because the chosen CBA protocol is $CBA^\star$, which allows for staggered termination). These users will thus start working on the next round ahead of the others, without any problems.

5.5 Incentives and Disincentives

In Bitcoin, there are two main sources of envisaged rewards. The first is the reward offered for including a given payment P in a block. The second is the reward earned for solving the complex "cryptographic riddle" necessary to generate a valid block. In Algorand, the second form of rewards is not applicable. One may, however, reward the verifiers and leader of a round with a small percentage of the amounts of the payments included in the official payset they generate. Such rewards may be paid automatically, either in an *inflationary*, in a *budget-balanced* model, or in a combination thereof.[17]

Of course, one may also easily disincentivize maliciousness by automatically imposing fines to users who provably deviate from their prescribed instructions in the protocol.[18]

6 Improvements

6.1 Optimizing Throughput

Maurice Herlihy has pointed out that throughput in Algorand can be significantly improved by *pipelining*.

[17] Automatically paying a reward w_X to a user X in a round r, essentially means that, in status S^r, the amount w_X is added to the amount $a_X^{(r)}$ that X would have had without this reward.

In an inflationary model, such reward w_X is not offset by any reduction in the amounts of money owned by other users. In a budget-balanced model, the reward w_X may, for instance, come from the payer Y of a given payment $P \in PAY^r$, in which case, in the status S^r, $a_X^{(r)}$ is automatically increased by w_X, while $a_Y^{(r)}$ is automatically decreased by w_X.

[18] For instance, assume that a malicious leader ℓ^r, in Message Step 2 on $\text{ROUND}(r)$, sends a digitally signed pair to one verifier, and a different digitally signed pair to another verifier. Then, exposing these contradictory signatures (e.g., by including them in the output of the CBA protocol) may automatically trigger a fine for ℓ^r. The fine may be as draconian as confiscating all the money ℓ owns in the system. This is a type of proof of stake [6].

6.2 Removing the Longer Corruption Time Assumption

In our description so far, the verifiers of a round r are *openly* selected from some previous quantity Q^{r-1}. That is, once Q^{r-1} becomes known close to round r (and only close to round r, so as to preserve the unpredictability of the verifiers as much as possible), every user can compute the entire verifier set TV^r.

We wish to point out, however, the existence of an alternative class of selection mechanisms, which we call *secretive*. In these mechanisms, an honest user X privately learns in round $r-1$ whether he has become a verifier for round r and, if this is the case, provably reveals that he is a round-r verifier to all users in round r. This shields the verifiers' identities until the last moment.

The Shielded-Verifier Version of Algorand. Continuing to temporarily assume that there is a one-to-one correspondence between keys and users in the system, in the basic version of Algorand, wemodify the verifier-selection mechanism as follows.

Let $p \in [0,1]$. Then,

A public key $X \in PK^{r-k}$ is selected as a verifier for round r if and only if
$$.H\left(s_X^r\right) \leq p,$$
where $s_X^r = SIG_X(r, X, Q^{r-1})$, and the quantity Q^{r-1} is that defined in ROUND(r).

Let us now argue that this mechanism, inspired by the micro-payment scheme of Micali and Rivest [26], essentially selects as round-r verifiers a fraction p of the users in PK^{r-k}.

The symbol "." that precedes the term "$H\left(s_X^r\right)$", in the middle line of the above italicized definition, is the *decimal point* used to mark the start of (in our case) binary expansion of a number between 0 and 1. Recall that $H(s_X^r)$ is a random 256-bit string. Accordingly, $.H(s_X^r)$ is the binary expansion of a random, 256-bit number in $[0,1]$, and thus is less than or equal to p with probability (essentially) p. Since the value s_X^r is uniquely determined by the public key X, in a way that is uncontrollable by X or anyone else, a key $X \in PK^{r-k}$ becomes a round-r verifier with probability p.

Because the value s_X^r always exists and is unique for each $X \in PK^{r-k}$, the verifier set TV^r so selected is indeed well defined. At the same time, TV^r is not publicly known. Indeed, for each $X \in PK^{r-k}$, s_X^r is unpredictable to anyone but X (when X is honest), and thus X is the only one to know whether he is a verifier for round r. However, *if he so wants*, he can convince all users that he is indeed a round-r verifier by propagating s_X^r.

We may refer to s_X^r as a *proof of membership* (or non membership) of X in TV^r. Indeed, upon receiving s_X^r, everyone can (a) verify that s_X^r is X's digital signature of (r, X, Q^{r-1}), (b) compute $H(s_X^r)$, and (c) verify that the so obtained result is less than or equal to p.

Each round-r verifier X is instructed to propagate his proof s_X^r only at the start of round r. Since malicious users, who have been secretly selected to be random-r verifiers, may not propagate their own own proofs of membership in TV^r, or may propagate them late, so that different honest users receive different sets of proofs of membership in TV^r, the verifier set TV^r is not publicly known to all users in the system. Rather, each user X knows his own subset, TV_X^r, of TV^r. We thus show that, despite this ambiguity, Algorand continues to work with this alternative, verifier-selection mechanism. There are a few details to consider. The first is leader selection.

Alternative Verifier Selection, Alternative Leader Selection. To ensure that in the above secretive selection mechanism the expected number of round-r verifiers is n, letting N be the

cardinality of PK^{r-k}, one sets $p = n/R$. (Of course, taking into account statistical fluctuations, one may want to choose a slightly larger p.)

Selecting p, however, only pins down —and only secretly at that— the set TV^r, but not a random ordering ρ^r of TV^r. Since in the basic version of Algorarnd the leader of round r, ℓ^r, was the first verifier in TV^r according to the ordering ρ^r, we need an alternative way to select ℓ^r, cognizant of the fact that the lexicographically first member of TV^r, and other straightforward ways, fail to work. One way that works is the following:

The leader of round r is the lexicographically first verifier X for which $H(s^r_X)$ is smallest.

This choice ensures that the leader of a round r, ℓ^r, is always well defined, but not that he will ever become publicly known. (Indeed, a malicious verifier Y may never propagate s^r_Y.) Let us thus informally argue that ℓ^r will be honest and publicly known with probability *at least* h, if the percentage of honest users is h. Recall that the value s^r_X is uniquely determined for each $X \in PK^{(r-k)}$. Then, since H is collision-resilient, with probability essentially 1 there is a bijection between the users X and the values $H(s^r_X)$. Since the latter values are random, their minimum will correspond to a random user in $PK^{(r-k)}$. That random user, W (for "winner"), will be honest with probability h. Being honest, W timely propagates his own proof of membership in TV^r, s^r_W, at the start of round r. Thus, all honest users will realize that W is the leader of round r. In fact, no malicious user M can manufacture a proof P' of membership in TV^r such that $P' < H(s^r_W)$, for the very good reason that, due to the guaranteed uniqueness of digital signatures, no such proof P' exists. Nor could the malicious users confuse the honest ones about the identity of the leader of round r, when the winner W is indeed honest, by propagating their proof of membership in TV^r purposely late. In fact their proofs will not yank W from his "winning position". That is, whether or not two honest players X and Y see the same set of proof of membership in TV^r, they will both see s^r_W, and none of them can possibly see a proof P such that $H(P) < H(s^r_W)$. Accordingly, both of them will recognize W as the leader of round r.

Note, however, that, if they so wanted(!), the malicious users might provide additional chances for an honest user to become the leader of round r.[19] In sum,

The leader ℓ^r of a round r is honest with probability $\geq h$.

Above, probabilities are taken not only over the random choices of the oracle H, but also over the possible coin tosses of all users (in this case of all malicious users, since the honest users are always asked to act deterministically in all our protocols).

In sum,

With the above secretive selection mechanism, the system has inclusiveness h.

Once again, such inclusiveness can be increased by selecting 3 or more leaders for each round. (The verifiers can use the leaders' majority opinion in order to select their own initial values in the CBA protocol.)

[19] For example, assume that the smallest value of the type $H(s^r_X)$ is $H(s^r_M)$, where M is malicious and does not propagate his proof of membership in TV^r in round r. Then, although M should have been the leader of round r, his "ducking" may cause a honest user to become the round leader.

A Few Additional Details. One may worry what may happen when the winner W is malicious. A malicious W may delay the propagation of his proof of membership, so that it is received only by some of the honest users, probably causing different honest users to have different opinions about who the leader of round r is.[20] It is easy to realize, however, that, even in this situation, Algorand's blockchain will, with overwhelming probability, *not fork*. In fact, the CBA protocol produces a unique certified output, and thus a unique official payset, whenever it is run with the required majority of honest verifiers, whether or not the honest verifiers agree on who the round leader is,[21] and whether or not they agree on the verifier set TV^r.[22]

By properly choosing the probability p, based on the percentage of honest users h, the percentage of honest player $h' < h$ required by the CBA protocol, and the chosen acceptable failure probability F, one can guarantee that the honest verifiers in TV^r will have the required majority with probability $> 1 - F$.

Of course, in a version of Algorand with a secretive verifier-selection mechanism, the certificate for the output of the CBA protocol at round r should, in addition to the digital signatures of a sufficient number of verifiers, also include the proof of membership of these verifiers. (At the same time, taking into account the possible fluctuation of the cardinality of TV^r, the number of sufficient signatures should be chosen so that the probability that the malicious verifiers could construct a valid looking certificate (and thus a fork) is negligible.

Advantages of the Shielded-Verifiers Version of Algorand. Implementing Algorand with a secretive selection mechanism adds some extra complexities, but also has a main advantage: it dispenses with the Longer Corruption Time assumption. That is, this new version of Algorand no longer requires to assume that the Adversary needs at least a few rounds to corrupt an honest user. Thus, a secretive verifier-selection mechanism *totally hides* the honest round-r verifiers from the Adversary until the very start of round r.

Thus, to cause a fork in this version of Algorand, the Adversary must be able not only to coordinate perfectly all the users he corrupts, but also to corrupt any user pretty much instantaneously (i.e., within a single round).

6.3 Removing the Continual Participation Requirement

An honest user in Algorand follows all his prescribed instructions, which certainly include participating to the protocol.

In the basic and last version of Algorand, a user missing to participate in even a single round is pessimistically judged malicious —although, in reality, he may have only experienced a network-connection problem, or simply taken a "break". (This harsh judgement was not capricious, but necessary. In both versions, in fact, the verifier set of a given round r was summoned, whether

[20] Needless to say that is different from planning to have multiple verifiers, in order to increase inclusiveness, as discussed above. Here, each honest verifier X believes there is a single verifier, ℓ_X^r, but for another honest verifier$Y$ we may have $\ell_X^r \neq \ell_Y^r$.

[21] In fact, a round-r verifier X runs the CBA protocol with an initial value v_X, and the agreement and consistency properties of Byzantine agreement hold, no matter how each v_X was selected.

[22] In fact, no matter what the malicious users might do, for each honest round-r verifier X, TV_X^r is a subset of TV^r, and the difference set $TV^r \setminus TV_X^r$ solely consist of malicious players. Thus, if the honest verifiers are in the required majority in TV^r, then an execution of CBA with "individual verifier sets" TV_X^r is, *in practice*, "equivalent" to an execution of CBA in which (1) the set of players is common knowledge and consists of TV^r, and (2) for every honest verifier X, the malicious players in TV^r but not in TV_X^r do not send any messages to X.

openly or secretly, in round $r-1$. If the chosen verifiers could not respond to the call and take on their duties because "absent", then the inclusiveness of both versions would be 0.)

Continual participation can be strongly expected in a "permissioned" system, where users are presumably carefully vetted before being admitted, but not in a permissionless system, despite all incentives offered.

What can be done?

One possibility would be to revise the current Honest Majority of Users assumption so as it applies only to the "currently active" users rather than the "currently existing" users. The meaningfulness of such a revised assumption, however, would dubious at best. Indeed, since "maliciousness never sleeps", malicious users may find themselves in the majority, if honest users were allowed to rest.

A better alternative is to modify the current version of Algorand so as to work with *"lazy-but-honest"* users. Roughly, these are users who honestly follow all their prescribed instructions when they participate to the protocol, but are asked to participate to the protocol only very rarely (e.g., once a month) and with advance notice. In such a case, it is much more reasonable to assume that an honest user will indeed *always participate when he is asked to.*

This is actually quite easy to achieve.

The Lazy-but-Honest Version of Algorand. In the Shielded-Verifiers version of Algorand, we modify the verifier-selection mechanism as follows.

A public key $X \in PK^{r-K-k}$ is selected as a verifier for round r if and only if $.H\left(s_X^r\right) \leq p$,
where $s_X^r = SIG_X(r, X, Q^{r-K})$ and K is much larger than k.

Notice that in this version of Algorand, an honest user X can compute the digital signature s_X^r only in round $r-K$, because only in that round will he learn the quantity $Q^{(r-K)}$. As in the previous versions of Algorand, however, the round-r verifiers are chosen among the users already present k rounds before that in which s_X^r is computed (so as to avoid that the malicious players, after learning the relevant digital signatures, might be able to manipulate the candidate set from which TV^r is chosen by introducing new public keys).

Importantly, a user learns whether he has become a round-r verifier way before round r: indeed K rounds before. Accordingly, accounting for the variability of the length of a round, a lazy-but-honest user X may be asked to go "online" every —say— $K/2$ rounds, so as to (1) retrieve the latest $K/2$ "Q quantities" produced since the last time he was online, and (2) generate his proper $K/2$ digital signatures of each one of them, in order to learn if he is going to be selected as a verifier in any of the next $K/2$ rounds.

If not, he can go off-line with an "honest conscience". Had he continuously participated, he would have essentially taken 0 steps in the next $K/2$ rounds, which is exactly what he is doing.

If yes, he will prepare himself by readying all the information —e.g., the status information— he needs to act honestly as a verifier at the proper round, and to earn the rewards he is entitled to. In a sense, the lazy-but-honest version of Algorand enables lazy users to remain honest. (In Appendix E we discuss efficient ways to learn the current status information securely.)

By so acting, such a user X only misses participating to the the propagation protocol. But a propagation protocol is typically robust and relies on who announces to be active anyway. (At most, to handle propagation with fewer active honest users, one has to use a larger number of "neighbors" and/or revise upwards the delivery time upperbound π.)

Notice that, if unpredictability were not a desideratum, then one could handle "lazy-but-honest" users by means of simpler and open selection mechanisms.[23]

6.4 Removing the One-Key-Per-User Assumption

So far, we have assumed that the public keys in each PK^r are in a one-to-one correspondence with the users in the system, an assumption that is hard to enforce, at least in a permissionless system.

Allowing users to generate multiple keys would enable them to enjoy the (pseudo) anonymity they enjoy in Bitcoin, but would also greatly endanger the security of the system. Indeed, by letting each malicious user own a great number of keys, the Adversary could ensure (a) that the leader of most rounds is malicious, thereby causing the inclusiveness of the system to be negligible, and (b) that most verifier sets have a malicious majority, thereby forcing the Algorand's blockchain to fork.

To avoid these security problems we put forward the *preferred version of Algorand*. This version essentially coincides with the lazy-but-honest one, after modifying the mechanisms for selecting the leader and the verifiers of a round, and relies on the Honest Majority of Money assumption.

The Preferred Verifier-Selection Mechanism At a high level, we take a proof-of-stake approach in selecting the verifier set TV^r. That is, we no longer assign to each key $X \in PK^{r-K-k}$ the same probability of being selected. Instead, we let the probability of choosing such X be proportional to a_X^{r-K-k}, that is, to the amount of money that X holds in round $r-K-k$. Let us explain how.

Assume that, in the lazy-but-honest version of Algorand, the (expected) number of verifiers we like to select is, for concreteness, 1,000; and that the total amount of money owned, in round $r-1$, by the keys in PK^{r-K-k} is, for concreteness again, 10^9. Then, we let a unit of weight correspond to an amount of money of 10^6.

Let $X \in PK^{r-K-k}$ and let the amount X owns in round $r-1$ be, for final concreteness, 3.7 millions. Then, to determine whether X becomes a verifier in round r, and the weight with which he becomes a verifier, we independently roll 4 biased coins: the first 3 have probability $1/10^6$ of coming up Heads; the forth has probability $.7/10^6$. Key X becomes a round-r verifier if at least one coin toss ends up Heads, and with weight w, $w \in \{1,2,3,4\}$, if w of the 4 coin tosses are Heads. To implement these 4 coin tosses, X first computes $s_X^r = SIG_X\left(X, r, H\left(Q^{r-1}\right)\right)$; second it computes the four values

$$v_{1,X} = H(1, s_X^r) \quad v_{2,X} = H(2, s_X^r) \quad v_{3,X} = H(3, s_X^r) \quad v_{4,X} = H(4, s_X^r);$$

third it computes the ith coin toss $c_{i,X}$, for $i = 1, ..., 3$, to be 1 if and only if $.H(i, s_X^r) < 1/10^6$; and lastly it computes the fourth coin toss $c_{4,X}$ to be 1 if and only if $.H(i, s_X^r) < .7/10^6$.

Accordingly, X is a round-r verifier if and only if $\sum_i c_i \geq 1$.

If this is the case, then X propagates s_X^r.

Note that such s_X^r is a proof that $X \in TV^r$, and that X's weight is $w_X = \sum_i c_i$. (If so wanted, we could augment this proof to include the actual values $v_{i,X}$ for which $c_i = 1$.)

[23] E.g., by the following one: *A public key $X \in PK^{r-K-k}$ is selected as a verifier for round r if and only if $.H(r, X, Q^{r-K}) \leq p$.* Algorand could also work with such " Open Self Selection" mechanisms.

The Preferred Leader Selection Mechanism A key $X \in PK^{r-K-k}$ is selected as the leader of round r with a probability that is again proportional to the money that X owns in round $r-1$. Namely,

Every user U personally recognizes a user $Z \in PK^{r-K-k}$ to be the leader of round r if and only if

(a) *U has received a proof s^r_Z that $Z \in TV^r$ and*

(b) *there exists an integer i such that $v_{i,Z} < v_{j,X}$*
for all integers j and all users X for which U has received a proof s^r_X that $X \in TV^r$.

Note that with probability at least 1/2, under the Honest Majority of Money assumption, every honest user personally recognizes the same user Z to be the leader of round r.

Once again, this implies inclusiveness 1/2, and as before, it does not matter that, in some rounds, different honest users may recognize different leaders.

A Few Additional Details After choosing the leader and the verifiers (and their weights) of each round in this fashion, the protocol continues "as before", with the same terminology even, after interpreting a verifier X with weight w_X as w_X "incarnations" of X, $X.1, \ldots, X.w_X$, and treat each incarnation as a separate user. For instance, "a set of at least k round-r verifiers" means "a subset K of TV^r such that the the sum of the weights of its members is at least k".

Switching Honesty Assumption

Note that, to remove the One-Key-Per-User assumption, we no longer rely on the Honest Majority of User assumption, but on Honest Majority of Money.

Gained Advantage A main advantage of assigning power to a key in proportion to the amount of money it owns is that a malicious user M no longer has any incentives to increase the number of keys he has in the system in order to gain more control. His influence over the system remains the same if —say— he has one million of monetary units in a single key, or a single monetary unit in one million keys. Should a malicious user have more power in the latter scenario, the set of keys would become so unwieldy as to cause the collapse of the system.

By contrast, in the present modification, a user may wish to spread his money over more keys only for privacy reasons, that is, to enjoy additional (pseudo) anonymity.

6.5 More General and Efficient Block Structures

We have developed a more general and flexible way to organize blocks of information. Our new *block structures* can replace blockchains in Algorand, and in any blockchain-based system, so as to enable one to prove very efficiently the content of an individual past block.

Recall that, to form a block chain, a block B_r has the following high-level structure:

$$B_r = (r, INFO_r, H(B_{r-1})) \ ,$$

where $INFO_r$ is the information that one wishes to secure within the rth block: in the case of Algorand, $INFO_r = PAY^r$.[24]

A well known fundamental property of block chains, recalled below in order to establish a common terminology and an abridged line of reasoning, is that no one can alter a past block without also changing the last block. Indeed, let the latest block be B_{last} and assume that one replaces B_r with a different block $\widetilde{B_r}$. Then, since H is collision resilient, $H(\widetilde{B_r})$ is —with overwhelming probability— different from $H(B_r)$. Accordingly, no matter how one chooses the information $\widetilde{INFO_{r+1}}$, for the block following $\widetilde{B_r}$ in the blockchain, namely $\widetilde{B_{r+1}} = (r+1, \widetilde{INFO_{r+1}}, H(\widetilde{B_r}))$, due again to the collision resiliency of H, it is the case that

$$\widetilde{B_{r+1}} \neq B_{r+1} \ .$$

This inequality propagates. That is, $\widetilde{B_{r+2}}$ differs from B_{r+2}; and so on, so that ultimately

$$\widetilde{B_{last}} \neq B_{last} \ .$$

This property enables one to verify the content of an individual past block, but only inefficiently.

Inefficient Verifiability of Individual Blocks in Blockchains Consider a person X who does not know the entire blockchain, but knows that B_z is a correct block in it. Then, the above fundamental property enables one to prove to such a person that any individual block B_r, where r is smaller than z, is also correct. Namely, one provides, as a "proof", all the intermediate blocks $B_{r+1}, \ldots, B_{z-1}$ to X, who then uses H to regenerate the blockchain from r onwards, until he reconstructs the zth block and checks whether or not it coincides with the block B_z he knows. If this is the case, then X is convinced of the correctness of B_r. Indeed, anyone capable of finding such a seemingly legitimate proof must also have found a collision in the hash function H, which is practically impossible to find.

To see the usefulness of such verifiability, consider the following example. Let the blockchain be generated by a payment system such as Bitcoin or Algorand , and let X be a judge in a court case in which the defendant wishes to prove to X that he had indeed made a disputed payment P to the plaintiff two years before. Since it it reasonable to assume that the judge can obtain the correct last block in the chain, or at least a sufficiently recent correct block, B_z, "all" the defendant has to do is to provide the proof $B_{r+1}, \ldots, B_{z-1}$ to the judge, who then verifies it as explained.

The problem, of course, is that such a proof is quite long. If each block comprises the payments of a one-minute interval, as in Algorand , then the proof consists of one million blocks. This is not a trivial amount of information. If there are roughly 1,000 payments per block, and each payment consists of 300 bytes (as is the case for Bitcoin), then the proof comprises 300 giga bytes.

[24] Recall that we use superscripts to indicate rounds in Algorand. This section, however, is dedicated to blockchains in general, and thus the rth block may not correspond to the rth round in the sense of Algorand. That is, above "r" is just the block number, and it is included in block B_r for clarity.

Also note that the above general structure of a block is conceptual. For instance, in Bitcoin B_r may include the digital signature of the block constructor, the one who has solved the corresponding computational riddle. In Algorand , the authentication of B_r —that is, a matching certificate $CERT_X$— may be provided separately. However, it could also be provided as an integral part of B_r. In this case, since there may be many valid certificates, the leader ℓ_r of round r may include, in its message to all round-r verifiers, also a valid certificate for the output of the previous round, so that agreement will also be reached on what the certificate of each round r is.

Blocktrees Since the ability to prove efficiently the exact content of a past individual block is quite fundamental, we develop new block structures.

In our new structures, like in blockchains, the integrity of an entire block sequence is compactly guaranteed by a much shorter value v. This value is not the last block in the sequence. Yet, the fundamental property of blockchains is maintained: any change in one of the blocks will cause a change in v.

The advantage of the new structures is that, given v, letting n be the number of blocks currently in the sequence, the content of each individual block can be proved much more efficiently. For instance, in *blocktrees*, a specific embodiment of our new block structures, each block can be proved by providing just $32 \cdot \lceil \log n \rceil$ bytes of information.

This is indeed a very compact proof. In a system generating one block per minute, then, after 2 millennia of operations, $\lceil \log n \rceil < 30$. Thus, less than 1KB —one kilo byte, that is, 1,000 bytes— suffice to prove the content of each individual block. (Less than 2KB suffice after two billions of years, and 4KB suffice essentially for ever.) Moreover, such a proof is verified very efficiently.

Details about blocktrees can be found in Appendix D.

6.6 Using More Sophisticated Cryptographic Tools

Algorand can also benefit from more sophisticated cryptographic tools. In particular,

1. *Combinable Signatures.* Often, in Algorand , some piece of data D must be digitally signed by multiple parties. To generate a more compact authenticated record, one can use combinable digital signatures. In such signatures, multiple public keys —e.g., PK_1, PK_2 and PK_3 could be combined into a single public key $PK = PK_{1,2,3}$— and signatures of the same data D relative to different public keys can be combined into a single signature relative to the corresponding combined public key. For instance, $SIG_1(D)$, $SIG_2(D)$ and $SIG_3(D)$ could be transformed into a single digital signature $s = SIG_{1,2,3}(D)$, which can be verified by anyone relative to public key $PK_{1,2,3}$. A compact record of the identifiers of the relevant public key, in our example the set $\{1, 2, 3\}$, can accompany s, so that anyone can quickly gather PK_1, PK_2 and PK_3, compute $PK = PK_{1,2,3}$, and verify the signature s of D based on PK.

   This allows to turn multiple related propagations into a single propagation. In essence, assume that, during a propagation protocol, a user has received $SIG_{1,2,3}(D)$ together with a record $\{1, 2, 3\}$ as well as $SIG_{4,5}(D)$ together with a record $\{4, 5\}$. Then he might as well propagate $SIG_{1,2,3,4,5}(D)$ and the record $\{1, 2, 3, 4, 5\}$.

2. *Tree-Hash-and-Sign.* When a signature authenticates multiple pieces of data, it may be useful to be able to extract just a signature of a single piece of data, rather than having to keep or send the entire list of signed items. For instance, a player may wish to keep an authenticated record of a given payment $P \in PAY^r$ rather than the entire authenticated PAY^r. To this end, we can first generate a Merkle tree storing each payment $P \in PAY^r$ in a separate leaf, and then digitally sign the root. This signature, together with item P and its authenticating path, is an alternative signature of essentially P alone.

3. *Certified Email.* One advantage of the latter way of proceeding is that a player can send his payment to ℓ by certified email,[25] preferably in a sender-anonymous way, so as to obtain a

[25] E.g., by the light-weight certified email of US Patent 5,666,420.

receipt that may help punish ℓ if it purposely decides not to include some of those payments in PAY_ℓ^r.

7 Permissioned Algorand

This version of Algorand balances privacy and traceability. It also provides a new, and not controlling, role for banks or other external entities. The version relies on the classical notion recalled below.

7.1 Digital Certificates

Assume that the public key PK of a party R, who has the authority to register users in the system, is publicly known. Then, after identifying a user X and verifying that a public key PK_X really belongs to X, R may issue a *digital certificate*, C_X, guaranteeing that not only PK_X is a legitimate public key in the system, but also that some suitable additional information I holds about X: in essence,

$$C_X = SIG_R(PK_X, I).$$

For instance, I may specify X's role in the system, the date of issuance of the certificate, a date of expiration of the certificate (i.e., the date after which one should no longer rely on C_X), etc. For instance, $C_X = SIG_R(PK_X, X, user, issued : 03/21/2016, expiring : 03/21/2017)$.[26]

Accordingly, by presenting his own digital signatures of a message m, $SIG_X(m)$, relative to a key PK_X, together with such C_X, X enables another party X who knows PK_R, and can thus verify C_X, to verify that $SIG_X(m)$ is indeed X's digital signature of a message m.

Digital certificates can also be chained. In a length-two chain, R may issue a first certificate to another entity R', specifying that R' has the authority to issue certificates, and then R' may issue a separate certificate for PK_X, so that C_X may be taken to consist of both these certificates.

Using Certificates to Prevent Illegal Activities

The payer and the payee of a payment made via a traditional check are readily identifiable by every one holding the check. Accordingly, checks are rarely used for money laundering or other illegal activities. Digital certificates, issued by a proper registration agent, could be used in Algorand to ensure that only some given entities can identify the owner X of a given public key PK_X, and only under proper circumstances. Let us give just a quick example.

There may be multiple registration authorities in the system. For concreteness only, let them be *banks*, whose public keys are universally known (or have been certified, possibly via a certificate chain) by another higher authority whose public key is universally known, and let G be a special entity, referred to as the *government*. To join the system a user X needs a bank-certified public key. To obtain it, he generates his own public-secret signature pair (PK_X, SK_X), and asks a bank B in the system to issue a certificate $C_X = SIG_B(B, PK_X, I)$. To be valid, a payment should always include a certificate for each of its payer and payee. In order to issue C_X, B is additionally

[26] If an expiration date is included, then, as long as x continues in good standing, R can automatically re-issue his certificate for the next time period. So, a certificate can very well have —say— only a single day of validity.

required to identify X, so as to produce some identifying information I_X.[27] Then, B computes $H(I_X)$, and makes it a (preferably) separate field of the certificate. That is,

$$C_X = SIB_B(B, PK_X, I, H(I_x)).$$

Since H is a random oracle, only the bank knows that PK_X's owner is X. However, if G wishes to investigate the payer or the payee of a given payment, it retrieves the relevant certificate C_X and the bank B that has issued C_X, and then asks or compels B, with proper authorization (e.g., a court order), to produce the correct identifying information I_X of X.

Note that the bank cannot reveal an identifying piece of information I'_X that is different from that originally inserted in the certificate, since H is collision-resilient.

It is also possible to insert in the certificate, instead of $H(I_X)$, an encryption of I_X with a key known to the bank, or an encryption of I_X, $E(I_X)$, with a key known to G and only to G, as when G uses a public-key (preferably probabilistic) encryption scheme: in symbols,

$$C_X = SIB_B(B, PK_X, I, E(I_x)).$$

This way, the identities of the payers and the payees of all payments are transparent to G, without having to request the cooperation of any bank.

Let us stress that neither B nor G, once the certificate C_X has been issued, has any control over X's digital signatures, because only X knows the secret key corresponding to PK_X. In addition, neither one of them can understand the sensitive information $\mathcal{I}$ of a payment P that X makes, because $H(\mathcal{I})$, and not $E(\mathcal{I})$, is part of P. Finally, neither B nor G is involved in processing the payment P, only the randomly selected verifiers are. Finally, if a bank-issued certificate C_X has no expiration date, then no one, either the bank B that has issued it, nor the government G, can "prevent user X to have access to his own money". The government can only retrieve, directly or via a court order, the identity of the payer and the payee of a payment. Thus, the discussed law-enforcement concern is put to rest. Yet, except for B and G, X's owner continues to enjoy the same (pseudo) anonimity he enjoys in Bitcoin, and similar systems, relative to any one else: banks, merchants, users, etc.

This is a new role for a bank B, and a role that it can easily perform for its customers, because B already knows them very well, and because it typically interacts with them from time to time. By certifying a key X of one of its customers, B performs a simple but valuable service, for which it may be rewarded in several ways.

An additional advantage of using banks as registration authorities in a permissioned version of Algorand is that, by a proper authorization of their clients, money can be transferred from their traditional bank accounts to their keys in Algorand, at the same exchange rate, so that *de facto* Algorand may be used as a very distributed, convenient, and self-regulated payment system, based on a national currency, if so wanted. In this case, users may entrust their respective banks to take over their verification duties, or at least host their secret keys in their secure servers, and use the banks' faster data networks for communication. Incentives can be split between banks and users in any proportion. Actually, banks may have sufficient benefits from handling the users' money and may not need any incentives.

[27] In particular, such I_X may include X's name and address, a picture of X, the digital signature of X's consent —if it was digital—, or X can be photographed together with his signed consent, and the photo digitized for inclusion in I_X. For its own protection and that of X as well, the bank may also obtain and keep a signature of X testifying that I_X is indeed correct.

Rewards from Retailers Only Whether or not the registration authorities are banks, and whether or not law-enforcement concerns are addressed, a permissioned deployment of Algorand enables one to identify (e.g., within its certificate C_X) that a given key X belongs to a *merchant*.

A merchant M, who currently accepts credit card payments, has already accepted his having to pay transaction fees to the credit card companies. Such an M may thus prefer paying a 1% fee in Algorand to paying the typically higher transaction fee in the credit card system (not to mention the fact that he will greatly prefer to be paid within seconds, rather than days; to be paid in a less disputable way; etc.)

Accordingly, it is possible to arrange that all rewards in Algorand are a small percentage of the payments made to merchants only. Although such payments are strictly contained in the set of all payments, it is possible to leverage rewards from merchants only in order to incentivize the processing of all payments.[28]

[28]For instance, letting A' be the total amount paid to retails in the payments of PAY^r, one could compute a *maximum potential reward* R' (e.g., $R' = 1\%A'$). The leader and the verifiers, however, will not collectively receive the entire amount R', but only a fraction of R' that grows with the total number (or the total amount, or a combination thereof) of all payments in PAY^r. For example, keeping things very simple, if the total number of payments in PAY^r is m, then the actual reward that will be distributed will be $R'(1 - 1/m)$. As before, this can be done automatically by deducting a fraction $1\%(1 - 1/m)$ from the amount paid to each retailer, and partitioning this deducted amount among the leader and verifiers according to a chosen formula.

APPENDIX

A The Proof-of-Work Approach of Bitcoin

The purpose of this section is not to describe accurately, or even with reasonable approximation, how Bitcoin works. Indeed, we shall leave out significant details (such as the organization of money in "individual coins", the incentive structure, etc.) in order to provide a flavor of the proof-of-work approach of Bitcoin, and thus to explain the three problems discussed in Section 2 arise.

In Bitcoin and its variants [12, 13, 24], a payment P is propagated in a peer-to-peer fashion. As new payments are continually made, at every point in time different users may indeed see different payments, and thus hold different views of the status of the system. Bitcoin addresses such possible confusion as follows.

Main Idea Bitcoin organizes the payments into a sequence of blocks, $B = B_1, B_2, \ldots$, so as to determine a *payment history* (i.e., the payments in B_1, followed by those in B_2, and so on) consisting of valid payments. Bitcoin delegates the generation of a new block to all users, but ensures that, in this collective generation effort, only one user is expected to succeed in —say— a ten-minute period, so as to prevent the confusion that would arise by having multiple alternative blocks being generated almost simultaneously. To this end, in order to be valid, a block must also contain the solution to a separate (indeed, block-specific) cryptographic riddle, whose level of difficulty is chosen so that only one solution is expected to be found by anyone in the world in a ten-minute interval. Solving such a riddle requires a lot of computation.

Informal Details About Block Generation Each block B_i, in addition to payments and other information, also contains a value v_i that can be chosen arbitrarily. To be valid, a block B_i (in addition to containing valid payments, relative to the payment history so far) must be such that the string $H(B_i)$ ends in k 0s. Since H is a random oracle producing binary strings longer than k, the probability that $H(B_i)$ ends in k 0s is 2^{-k}. After preparing a (candidate) block B_i, a user hashes it to verify whether indeed $H(B_i)$ ends in k 0s. If this is not the case, the user may keep the same set of payments in the candidate block (or may add a few more that he has recently seen), *change the value* v_i, and hash the so changed B_i again, until he eventually succeeds, in which case he propagates the valid block B_i. For instance, if $k = 40$, then roughly a quadrillion attempts are expected before succeeding at forming a new valid block.

If the number of the users actively trying to form a new block is fixed, then it is possible to fix k so that the expected time to generate a new block is 10 minutes. When the number of such users varies over time, then it is necessary to adjust k accordingly.[29]

Multiple blockchains and the True Time to Payment in Bitcoin In Bitcoin, from the point of view of an individual user, the *public ledger* consists of the longest legitimate blockchain the user sees. Such a chain, however, may not be the same for all users, and a user may see the last few blocks of his longest blockchain totally change. Let us explain.

[29] For instance, if, while k stays the same, the number of users trying to generate a new block dramatically increases, then one notes that a new block may arise in 2 rather than 10 minutes. But as this fact is noticed, Bitcoin automatically re-adjusts k. This fact, based on our description so far, of course makes it more complex to verify whether a sequence of blocks is a legitimate block chain. But we do not need to delve in these complications.

Let X and Y be two different users who, seeing the same longest blockchain $B = B_1, \ldots, B_{i-1}$, almost simultaneously succeed to generate and append to it a new block. Since, however, the recent payments they see need not be the same, the new blocks they generate, respectively B_i^X and B_i^Y, may be different. So, for the moment, at least X and Y hold different opinions about the last block of "their own" public ledger.

At this point, each of X and Y propagates his own new block, while other users make and propagate their own new payments. During all this propagation, many (or even most) other users may thus see both the chain $B^X = B_1, \ldots, B_{i-1}, B_i^X$ and the chain $B^Y = B_1, \ldots, B_{i-1}, B_i^Y$. Let Z be such a user. Accordingly, when trying to create a new block, Z must choose which of the two chains to elongate. That is, when constructing a new candidate block, Z must choose whether to include in it $H\left(B_i^X\right)$ or $H\left(B_i^Y\right)$. Assume that Z generates and starts disseminating a new block, B_{i+1}^Z, elongating the chain B^Y, while the blocks B_i^X and B_i^Y are still being propagated. (Indeed, although a new block is expected to be produced in 10 minutes, it is possible to produce it in a much shorter time.) Then, for a while, it may happen that, in the eyes of at least some users, four public ledgers exist: namely, the $(i-1)$-block blockchain B, the i-block blockchains B^X and B^Y, and the $(i+1)$-block blockchain B^Z.

Because users are asked to elongate the longest blockchain they see, we expect that, from some point in time onwards, only one of B^X and B^Y will be a sub-chain of every user's public ledger. That is, we expect that the ith block of the public ledger will eventually stabilize. Although it is hard to predict when such stabilization may occur, in practice one may assume that the ith block of a blockchain that is at least $(i+6)$-block long will no longer change.

Accordingly, if a transaction, belonging to the seventh last block, transfers an amount a from public key PK to public key PK', then the owner of PK' 'can consider himself paid'. That is, in Bitcoin the true "time to payment" is not ten minutes, but one hour.

B Alternative Verifier-Selection Mechanisms

So far, Algorand selects the leader ℓ_r and the verifier set TV^r of a round r automatically, from quantities depending on previous rounds, making sure that TV^r has a prescribed honest majority. We wish to point out, however, alternative ways to select the verifiers and the leader.

One such way, of course, is via a cryptographic protocol (e.g., a multi-party function evaluation protocol in the sense of Goldreich, Micali, and Wigderson [**?**]) run by all users. This approach, however, is hopelessly slow when the number of users in the system is high. Let us thus consider two classes of alternative mechanisms: *chained* , *nature-based* , and *trusted-party* mechanisms.

One may also consider mixing the latter mechanisms with those already discussed.

Chained Mechanisms

Since each TV^r (which may include ℓ^r) has an honest majority, we could have TV^r itself (or more generally some of the verifiers of the rounds up to r) select the verifier set and/or the leader of round r. For instance, they could do so via multi-party secure computation. Assuming that the initial verifier set is chosen so as to have an honest majority, we rely on boot-strapping: that is, the honest majority of each verifier set implies the honest majority of the next one. Since a verifier set is small with respect to the set of all users, his members can implement this selection very quickly.

Again, it suffices for each round to select a sufficiently long random string, from which the verifier set and the leader are deterministically derived.

Nature-Based Mechanisms

The verifier set TV^r and the leader ℓ^r of a given round r can be selected, from a prescribed set of users PK^{r-k}, in a pre-determined manner from a random value v_r associated to the round r. In particular, v_r may be a *natural and public random value*. By this we mean that it is the widely available result of a random process that is hardly controllable by any given individual. For instance, v_r may consist of the temperatures of various cities at a given time (e.g., at the start of round r, or at a given time of the previous round), or the numbers of stock of given security traded at a given time at a given stock exchange, and so on.

Since natural and public random value may not be sufficiently long, rather than setting

$$TV^r \stackrel{v_r}{\leftarrow} PK^{r-k} \ ,$$

we may instead set

$$TV^r \stackrel{H(v_r)}{\leftarrow} PK^{r-k} \ .$$

Trustee-Based Mechanisms

An alternative approach to selecting TV^r involves one or more distinguished entities, the *trustees*, selected so as to guarantee that at least one of them is honest. The trustees may not get involved with building the payset PAY^r, but may choose the verifier set TV^r and/or the leader ℓ^r.

The simplest trustee-based mechanisms, of course, are the single-trustee ones. When there is only one trustee, T, he is necessarily honest. Accordingly, he can trivially select, digitally sign, and make available TV^r (or a sufficiently random string s_r from which TV^r is derived) at round r.

This a simple mechanism, however, puts so much trust on T. To trust him to a lesser extent, T may make available a single string, s_r, uniquely determined by the round r, that only he can produce: for instance, $s_r = SIG_T(r)$. Then, every user can compute the random string $H(SIG_T(v_r))$, from which TV^r is derived.

This way, T does not have the power to control the set TV^r. Essentially, he has a single strategic decision at his disposal: making $SIG_T(r)$ available or not. Accordingly, it is easier to check whether T is acting honestly, and thus to ensure that he does so, with proper incentives or punishments.

The problem of this approach is unpredictability. Indeed, T may compute $SIG_T(r)$ way in advance, and secretly reveal it to someone, who thus knows the verifier set of a future round, and has sufficient time to attack or corrupt its members.

To avoid this problem, we may rely on secure hardware. Essentially, we may have T be a tamper-proof device, having a public key posted "outside" and a matching secret key locked "inside", together with the program that outputs the proper digital signatures at the proper rounds. This approach, of course, requires trusting that the program deployed inside the secure hardware has no secret instructions to divulge future signatures in advance.

A different approach is using a natural public random value v_r associated to each round r. For instance, T may be asked to make available $SIG_T(v_r)$. This way, since the value v_r of future rounds r is not known to anyone, T has no digital signature to divulge in advance.

The only thing that T may still divulge, however, is its own secret signing key. To counter this potential problem we can rely on k trustees. If they could be chosen so as to ensure that a suitable majority of them are honest, then they can certainly use multi-party secure computation to choose TV^r at each round r. More simply, and with less trust, at each round r, we may have each trustee i make available a single string, uniquely associated to r and that only i can produce,

and then compute TV^r for all such strings. For instance, each trustee i could make available the string $SIG_i(r)$, so that one can compute the random string $s^r = H(SIG_1(r), \ldots, SIG_k(r))$, from which the verifier set TV^r is derived. In this approach we might rely on incentives and punishments to ensure that each digital signature $SIG_i(r)$ is produced, and rely on the honesty of even a single trustee i to ensure that the sequence $s^1, s^2, \ldots$ remains unpredictable.

The tricky part, of course, is making a required string "available". If we relied on propagation protocols then a malicious trustee may start propagating it deliberately late in order to generate confusion. So, trustee-based mechanisms must rely on the existence of a "guaranteed broadcast channel", that is, a way to send messages so that, if one user receives a message m, then he is guaranteed that everyone else receives the same m.

Finally, rather than using secure computation at each round, one can use a secure computation pre-processing step. This step is taken at the start of the system, by a set of trustees, selected so as to have an honest majority. This step, possibly by multiple stages of computation, produces a public value pv and a secret value v_i for each trustee i. While this initial computation may take some time, the computation required at each round r could be trivial. For instance, for each round r, each trustee i, using his secret value v_i, produces and propagates a (preferably digitally signed) single reconstruction string s_i^r, such that, given any set of strings $\mathcal{S}^r$ that contains a majority of the correct reconstruction strings, anyone can unambiguously construct TV^r (or a random value from which TV^r is derived). The danger of this approach, of course, is that a fixed set of trustees can be more easily attacked or corrupted.

C Certified Byzantine Agreement

C.1 The Notion of Certified Byzantine Agreement

Recall that, in a protocol $\mathcal{P}$, a player is *honest* if, in every execution of $\mathcal{P}$, he follows all his prescribed instructions, and *malicious* otherwise. An execution of $\mathcal{P}$ terminates when each honest player terminates. In an execution of $\mathcal{P}$, malicious players are allowed to deviate from their instructions in every way they want, and even to perfectly coordinate their actions.

Let us start by recalling the simpler, traditional notion of a Byzantine agreement. This notion was introduced by Pease Shostak and Lamport [14] for the binary case, that is, when every initial value consists of a bit. However, it was quickly extended to allow for arbitrary initial values. (See the surveys of Fischer [15] and Chor and Dwork [16].)

Definition *(Traditional Byzantine Agreement)* Let $\mathcal{P}$ be a protocol in which every player X has an initial value v_X and, if he terminates, outputs a value OUT_X. Then, $\mathcal{P}$ is a *Byzantine agreement protocol* tolerating t malicious players if, in every execution in which at most t players are malicious, it terminates with the following guarantees:

1. *Agreement:* $OUT_X = OUT_Y$ for all honest players X and Y.
2. *Consistency:* if the inputs of all honest players coincide with some value v, then $OUT_X = v$ for every honest player X.

A Byzantine agreement protocol has fault tolerance f, $f \in [0, 1]$, if it tolerates a fraction f of the actual number of players. $\triangle$

Recall that Algorand needs a subroutine guaranteeing not only that the n verifiers of a given round r reach consensus among themselves about PAY^r, but also that they can prove to all users what PAY^r is. Accordingly, it is not enough that they execute a Byzantine agreement protocol using as initial inputs the digitally signed value $PAY^r_{\ell^r}$ provided to each one of them by the leader ℓ^r. They need to run a stronger type of consensus protocol. Namely, one in which each (honest) user X outputs in addition to OUT_X also a certificate, $CERT_X$, enabling to convince all users, including those who have not participated to the consensus protocol, that OUT_X really is the output of a given execution. Such a certificate $CERT_X$ is close in spirit to a digital signature. That is, it is a string that can be quickly inspected for correctness (via a suitable verification algorithm VRF), but hard to forge. Slightly more precisely, recalling that n is the number of verifiers,

Definition *(Certified Byzantine Agreement)* Let $VRF : \mathbb{Z}^+ \times \{0,1\}^* \times \{0,1\}^* \to \{0,1\}$ be an efficiently computable function. Let $\mathcal{P}$ be a protocol in which, in an execution r, each player X has an initial value v^r_X and, upon termination, produces a value OUT^r_X and a string $CERT^r_X$. Then, $\mathcal{P}$ is a *certified Byzantine Agreement* (CBA) protocol with fault tolerance f and verification algorithm VRF if, for every execution r in which a fraction greater than f of the players are honest, with overwhelming probability $\mathcal{P}$ terminates satisfying the following properties:

1. *Agreement:* $OUT^r_X = OUT^r_Y$ for all honest verifiers X and Y.
2. *Consistency:* if, for some value v, $v^r_X = v$ for all honest players X, then $OUT^r_X = v$ for all honest players X.
3. *Certification.* For every honest player X
   - $VRF\,(r, OUT^r_X, CERT^r_X) = 1$, and
   - If $OUT^r_X \neq v$, then it is computationally intractable to produce any string $FAKE$ such that $VRF\,(r, v, FAKE) = 1$.

C.2 The Protocol *Classical* CBA

Classical CBA is a trivial adaptation of the protocol of Dolev and Strong [17], and is thus omitted.

This protocol has fault tolerance $1/2$ and requires $t+1$ stages of n-to-n communication, if there are t malicious players.

C.3 The Protocol $CBA^\star$

Protocol $CBA^\star$ has fault tolerance $2/3$.

We construct $CBA^\star$ based on

(a) the reduction of Turpin and Coan [18], showing that an arbitrary-value Byzantine agreement protocol can be constructed by adding just two stages of n-to-n communication to any binary Byzantine agreement protocol, and

(b) the randomized binary Byzantine agreement protocols of Feldman and Micali [22] (in turn based on that of Rabin [20] and Ben-Or [19]).

Essentially without modifications, the reduction of Turpin and Coan also shows that an arbitrary-value CBA protocol can be constructed by adding just two stages of n-to-n communication to any binary CBA protocol.

Let us thus provide the intuition for constructing a binary CBA protocol, *Binary* $CBA^\star$.

Intuition for *Binary* $CBA^\star$

Consider the following idealized protocol $\mathcal{P}$. This protocol consists of three phases, in which each player X starts with a bit b_X, which he then updates. The bit b_X represents X's current opinion about what the binary output of the protocol should be. Initially, b_X is the original binary input of X.

1. Every X sends b_X to all players, including himself.
2. A new randomly and independently selected bit c magically appears in the sky, visible to all.
3. Every player X updates b_X as follows.

   3.1 If X has just received 0 from more than 2/3 of the players, then he (re)sets b_X to 0.

   3.2 If X has just received 1 from more than 2/3 of the players, then he (re)sets b_X to 1.

   3.3 Else X (re)sets b_X to c.

We refer to each such bit c as a *common coin*.

Quick Analysis.

There are four possible cases to consider: namely, in the same phase 3 of an execution of $\mathcal{P}$,

`(i)` All honest players update their bit according to 3.1 or 3.2.

In this case, either all honest players update their bits according to 3.1, or all of them do so according to 3.2. Thus, agreement holds at the end of this phase: that is,

$$b_X = b_Y \text{ for all honest } X \text{ and } Y.$$

`(ii)` All honest players update their bits according to 3.3.

In this case agreement is reached right away: indeed, $b_X = c$ for all honest X.

`(iii)` Some honest players update their bits according to 3.1, and all others according to 3.3.

Let $\mathcal{H}_0$ (respectively, $\mathcal{H}(1)$) be the set of honest players updating their bits according to 3.1 (respectively, 3.2). If $c = 0$, then all players in $\mathcal{H}(0)$ reset their bits to 0, while all those in $\mathcal{H}(1)$ reset theirs to c, which is 0. Since $c = 0$ with probability 1/2, then in case `(iii)` agreement on 0 is reached with probability 1/2.

`(iv)` Some honest players update their opinion according to 3.2, and all others according to 3.3.

In this case, a symmetric argument shows that agreement is reached on 1 with probability 1/2.

In sum, at the end of each execution of the idealized protocol $\mathcal{P}$, no matter what malicious players might do, agreement is reached with probability at least 1/2. Moreover, in virtue of the update rules 3.1 and 3.2, once agreement is reached on some bit b, agreement continues to hold on the same bit b, even if protocol $\mathcal{P}$ is executed again and again. Finally, if all honest verifiers started with the same initial bit (i.e, if the honest players were originally in agreement on b), then they will continue to agree on b.

A problem with the above idealized protocol $\mathcal{P}$ is that, when agreement is reached, the honest verifiers are not aware that this is the case.

From Unaware Agreement to Aware Agreement. One way to avoid the above problem is to iteratively execute $\mathcal{P}$ —say— 100 times. Thus, the probability that agreement is not reached at the end of the $100th$ iteration is 2^{-100}, and can be safely ignored. This approach, however, wastes many iterations of $\mathcal{P}$, because the overwhelming majority of the times, agreement would have been reached way before the 100th iteration.

The following, still idealized, approach instead allows for fast and detectable termination. Rather than iterating independent executions of $\mathcal{P}$, iterate three correlated executions of $\mathcal{P}$.

The first one assumes that $c = 0$, that is, "forces the random bit c in the sky to be 0".

The second assumes that $c = 1$.

The third lets c be a randomly and independently selected bit.

As already mentioned, the third execution enables reaching agreement with probability 1/2. The first two instead ensure that, if agreement has already been reached on some bit b, an honest player X can learn that this is the case, and terminate the iteration with the binary output b. Let us see why this is the case.

Assume that agreement has just been reached on 0, let X be an honest player, and consider phase 1 of the next execution of $\mathcal{P}$ in which c is forced to be 0. Then, in that phase, X receives 0 from all honest players. Accordingly, no matter what bits the malicious players may send, X receives 0 from more than 2/3 of the players. Therefore, he can send 0 in three more phases and safely terminate outputting 0. Better yet, he terminates right away, because the predetermined rule is that, when one does not hear from X in a phase, he pretends that X says the same thing he said last. Indeed, when X so terminates, he knows that either there is already agreement on 0, or agreement will be reached on 0 at the end of the next phase 1.

By symmetry, if agreement has been just reached on 1, then in phase 2 of $\mathcal{P}$, where c is forced to be 1, an honest X can safely terminate outputting 1.

From common coins to Frequently Common Coins

As we have seen, each common coin c in phase 2 of $\mathcal{P}$ brings the verifiers into agreement with probability 1/2.

Notice, however, that it is not necessary to have a common coin c in each phase 2. Trivially, if the bit c were common in each phase 2, but random and independent only with probability 2/3, then such a magic bit would bring the verifiers into agreement with probability 2/3 at each phase 2. Slightly less trivially, the players would get into agreement with probability 2/3, at each phase 2, if, at each phase 2, each verifier X sees a bit c_X such that,

- with probability 2/3,

  (i) c_X is randomly and independently selected in $\{0, 1\}$, and

  (ii) $c_X = c_Y$, for all verifiers X and Y

(i.e., the "coin" happens to be *common*); and

- with complementary probability, the bits c_X are not only independent, but adversarially chosen (i.e., the "coin" is *partial*).

We call such a vector of bits $\{c_X : X \in TV^r\}$ a *partially common coin.*

The reason why such a coin brings the verifiers into agreement with probability 2/3 at each phase 2 is that, once the coin happens to be common, then it brings the verifiers into agreement, and, after they are in agreement, no matter what the bit vector $\{c_X : X \in TV^r\}$ might be, the honest verifiers, due to rules 3.1 and 3.2, remain in agreement.

Let us now show how to implement a partially common coin.

Binary $CBA^\star$ via Partially Commom Coin.

In order to turn the idealized binary CBA protocol $\mathcal{P}'$ into the binary CBA protocol $CBA^\star$, we

- implement, for each iteration i of $\mathcal{P}'$, a partially common coin $\{c_X^{(i)} : X \in TV^r\}$, and
- construct a suitable certificate $CERT_X$ whenever agreement has been reached.

In the context of our Round(r) protocol, Binary $CBA^\star$ is run in round r, where each user already has computed the already discussed quantity Q^{r-1}. Accordingly, we implement each coin c as follows.

1. In the first phase 2 of $\mathcal{P}$, each verifier $X \in TV^r$
   propagates the value $v_X \triangleq SIG_X(r, X, Q^{r-1}, 1)$;
   computes the smallest $M \in TV^r$ such that $H(v_M) \leq H(v_Y)$ for all $Y \in TV^r$; and
   sets $c_X^{(1)}$ to be the least significant bit of $H(v_M)$.
2. In the second phase 2 of $\mathcal{P}$, each verifier $X \in TV^r$
   propagates the value $v_X \triangleq SIG_X(r, X, Q^{r-1}, 2)$;
   computes the smallest $M \in TV^r$ such that $H(v_M) \leq H(v_Y)$ for all $Y \in TV^r$; and
   sets $c_X^{(2)}$ to be the least significant bit of $H(v_M)$.
3. And so on.

Note that all the $H(v_X)$ are random (because H is a random oracle) and independent (because for no two rounds and two verifiers are the inputs to H the same, since (r, X, i) is always retrievable from $SIG_x(r, X, i)$) 256-bit numbers). Thus the least significant bit of $H(v_M)$ is a random bit.

Let us now put it all together.

Description of Protocol $CBA^\star(r)$

Protocol $CBA^\star(r)$ is executed at round r —and is in fact called by the already described protocol Round(r)— when the set of verifiers TV^r and the quantities Q^{r-1} have already been computed.

The players of $CBA^\star(r)$ are the round-r verifiers. Each verifier i uses 4 local variables: x_i, y_i, z_i, holding either valid paysets or the value $\perp$, and a Boolean b_i. Each verifier also holds a counter γ, initially set to 1. Each message sent includes the round number in which it is sent as well as the

digital signature of its sender, although we omit these fields from the text for brevity. In each step, a verifier i can only send a single message to another verifier. If a step calls for a verifier i to send a message m to another verifier j, then m_j denotes the first message of i actually received by j.

Steps 3 through 5 constitute a loop, where the verifiers repeatedly exchange Boolean values, and different verifiers may exit this loop at different times. To exit this loop, a verifier V_i broadcasts a special value, denoted $0*$ (or $1*$), instructing the others to "pretend" they received 0 (or 1) from V_i in any subsequent Boolean exchanges.

D Blocktrees

We construct blocktrees, our block structure alternative to blockchains, by properly modifying a much older notion recalled below.

D.1 Merkle Trees

Merkle trees are a way to authenticate n already known values, $v_0, \ldots, v_{n-1}$, by means of a single value v, so that the authenticity of each value v_i can be individually and efficiently verified.

For simplicity, assume that n is a power of 2, $n = 2^k$, so that each value is uniquely identified by a separate k-bit string, s. Then, a Merkle tree T is conceptually constructed by storing specific values in a full binary tree of depth k, whose nodes have been uniquely named using the binary strings of length $\leq k$.

The root is named ε, the empty string. If an internal node is named s, then the left child of s is named $s0$ (i.e., the string obtaining by concatenating s with 0), and the right child is named $s1$. Then, identifying each integer $i \in \{0, \ldots, n-1\}$, with its binary k-bit expansion, with possible leading 0s, to construct the Merkle tree T, one stores each value v_i in leaf i. After this, he chooses the contents of all other nodes in a bottom up fashion (i.e., he chooses first the contents of all nodes of depth $k-1$, then those of all nodes of depth $k-2$, and so on). If v_{s0} and v_{s1} are respectively contained in the left and right child of node s, then he stores the 256-bit value $v_s \triangleq H(v_{s0}, v_{s1})$ in node s. At the end of this process, the root will contain the 256-bit value v_ε.

A Merkle tree of depth 3 is shown in Figure 1.A, at the end of this paper.

Assume now that v_ε is known or digitally signed, and let us show how each original value v_i can be authenticated relative to v_ε.

Consider the (shortest) path P that, starting from a node x, reaches the root. Then, the *authenticating path* of the content v_x of x is the sequence of the contents of the *siblings* of the nodes in P, where the sibling of a node $s0$ is node $s1$, and viceversa. Accordingly, an authenticating path of a leaf value in a tree of depth k consists of $k-1$ values. For example, in a Merkle tree of depth 3, the path from leaf 010 to the root is $P = 010, 01, 0, \varepsilon$, and the authenticating path of v_{010} is v_{011}, v_{00}, v_1, since the root has no sibling. See Figure 1.B at the end of this paper.

To verify (the authenticity of) v_i, given its authenticating path, relative to the root value v_ε, one reconstructs the contents of the nodes in the path from leaf i to the root, and then checks whether the last reconstructed value is indeed v_ε. That is, if the authenticating path is $x_1, \ldots, x_{k-1}$, then one first H-hashes together v_i and x_1, in the right order —i.e., computes $y_2 = H(v_i, x_1)$, if the last bit of i is 0, and $y_2 = H(x_1, v_i)$ otherwise. Then, one H-hashes together y_2 and x_2, in the right order. And so on, until one computes a value y_k and compares it with v_ε. The value v_i is authenticated if and only if $y_k = v_\varepsilon$.

The reason why such verification works is, once again, that H is collision resilient. Indeed, changing even a single bit of the value originally stored in a leaf or a node also changes, with overwhelming probability, the value stored in the parent. This change percolates all the way up, causing the value at the root to be different from the known value v_ε.

D.2 From Merkle Trees to Blocktrees

As we have seen, Merkle trees efficiently authenticate arbitrary, and arbitrarily many, *known* values by means of a single value. Indeed, in order to authenticate k values $v_0, \ldots, v_{k-1}$ by the single root content of a Merkle tree, one must first know $v_0, \ldots, v_{k-1}$ in order to store them in the first k leaves of the tree, store e in other proper nodes, and then compute the content of all other nodes in the tree, including the root value. Merkle trees have been used in Bitcoin to authenticate the payments of a given block. Indeed, when constructing (or attempting to construct) a given block, one has already chosen the payments to put in the block.

However, using Merkle trees to authenticate the block sequence as it grows is more challenging, because one does not know in advance what blocks to authenticate. Nonetheless, let us show that Merkle trees can, used in a novel way, yield new block structures enabling the efficient provability of individual blocks.

Conceptually, in our alternative structures, a block B_i has the following form:

$$B_r = (r, INFO_r, \mathcal{S}_r) \ ,$$

where the *structural information* $\mathcal{S}_r$ is a sequence of $\lceil \log r \rceil$ 256-bit strings, that is, $\lceil \log r \rceil$ strings of 32 bytes each. Below, we provide just a specific example of a suitable structural information $\mathcal{S}_r$, but it will be clear that our techniques enable myriads of efficient block structures alternatives to blockchains. Those corresponding to our specific choice of $\mathcal{S}_r$ are *block trees*.

Blocks in Blocktrees For brevity, let us set $INFO_r = v_r$. Conceptually speaking again, we start with a full binary tree T of depth k such that 2^k upper-bounds the number of possible values v_r. The values $v_0, v_1, \ldots$ are produced in order. When a new value v_i is generated, it is, again figuratively speaking, stored in leaf i of T, and then various strings are computed and stored in the nodes of T, so as to construct a Merkle tree T_i. One of these strings is the distinguished string e. When appearing in a node x of T_i, string e signifies that no descendant of x belongs to T_i.

When the first value, v_0, is generated and stored in leaf 0, T_0 coincides with (the so filled) node 0 of T. In fact, such T_0 is an elementary Merkle tree. Its depth is $\lceil \log(0+1) \rceil = 0$, its root is $R_0 = 0$, and it stores v_0 in its first depth-0 leaf (and in fact in its only leaf and node).

When the $i+1^{st}$ value, v_i, has been generated and stored in leaf i of T (possibly replacing the string e already there), the Merkle tree T_i is constructed as follows from the previous Merkle tree T_{i-1}. (By inductive hypothesis, T_{i-1} has depth is $\lceil \log i \rceil$; root R_{i-1}; and i depth-$\lceil \log(i+1) \rceil$ leaves, respectively storing the values $v_0, \ldots, v_{i-1}$.)

Let $R_i = R_{i-1}$, if leaf i is a descendant of R_{i-1}, and let R_i be the parent of R_{i-1} otherwise. Let P be the (shortest) path, in T, from leaf i to node R_i. For every node j in P, store the special string e in its sibling j', if j' is empty. Finally, for each node s in P, in order from leaf i (excluded) to node R_i (included), store in s the value $v_s = H(v_{s0}, v_{s1})$, if v_{s0} and v_{s1} respectively are the values stored in the left and right child of s. It is easy to see that the subtree of T rooted at R_i, storing the so computed values in its nodes, is a Merkle tree. This Merkle tree is T_i.

The construction of the first 8 consecutive Merkle trees, when the initially empty full binary tree T has depth 3, is synthesized in Figure 2, at the end of the paper. Specifically, each subfigure 2.i highlights the Merkle tree T_i by marking each of its nodes either with the special string e (signifying that "T_i is empty below that node"), or with a number $j \in \{0, \ldots, i-1\}$ (signifying that the content of the node was last changed when constructing the Merkle tree T_j). To highlight that the content of a node, lastly changed in T_j, will no longer change, no matter how many more Merkle trees we may construct, we write j in bold font.

With this in mind, we generate our block-tree structure as follows. After choosing the information $INFO_i$ that we want to secure in the ith block, we store the value $v_i = INFO_i$ into leaf i of T; construct the Merkle tree T_i; and set

$$\mathcal{S}_i = (R_i, auth_i) \ ,$$

where R_i is the root of T_i and $auth_i$ is the authenticating path of v_i in T_i. Then, the block is

$$B_i = (i, INFO_i, \mathcal{S}_i) \ .$$

Notice that $\mathcal{S}_i$ indeed consists of $\lceil \log i \rceil$ strings. To ensure that each string in $auth_i$, and thus every string in $\mathcal{S}_i$, is actually 256-bit long, rather than storing v_i in leaf i, we may store $H(v_i)$ instead.

Efficient Block Constructibility To construct the structural information $\mathcal{S}_i$ that is part of block B_i, it would seem that one would need information from all over the Merkle tree T_i. After all, $INFO_i$ and thus the value v_i stored in leaf i, are readily available, but the authenticating path of v_i, $auth_i$, comprises contents of nodes of previous trees, which in principle may not be readily available. If one had to obtain the entire T_{i-1} in order to construct $\mathcal{S}_i$, then constructing a new block B_i might not be too efficient.

However, note that, very much in the spirit of block chains, each B_i is trivially computable from the previous block B_{i-1} and the chosen information $INFO_i$. Indeed, each string in $\mathcal{S}_i$ is one of

(a) $H(INFO_i)$,

(b) the fixed string e,

(c) a string in $\mathcal{S}_{i-1}$, and

(d) a string obtained by hashing in a predetermined manner strings of the above types.

Figure 3, at the end of this paper, highlights —via a thick border— the nodes whose contents suffice to compute $\mathcal{S}_i = (R_i, auth_i)$ for the construction of the first 8 blocks in a blocktree. Specifically, each subfigure 3.i highlights the nodes whose contents suffice for generating $\mathcal{S}_i$. Each highlighted node is further marked a, b, or c, to indicate that it is of type (a), (b), or (c). Nodes of type (d), including the root R_i, are left unmarked.

In sum, in a blocktree-based system, block generation is very efficient.

Efficient Block Provability Let us now analyze proving the content of an arbitrary individual block B_r to someone knowing a subsequent block (e.g., the current last block).

Let $B_z = (z, INFO_z, \mathcal{S}_z)$ be a block, where $\mathcal{S}_z = (R_z, auth_z)$ and $z > r$. Then, to prove that B_i is the correct ith block relative to B_z, it suffices to provide the authenticating path of $INFO_i$ in the Merkle tree T_z.[30] Thus, such a proof comprises only $\lceil \log z \rceil$ 32-bit strings.

As already pointed out in Subsection 6.5, $\log n$ is at most 30 in most applications (and at most 100 in essentially all human applications). Thus, quite differently from the case of blockchains proving the content of an individual block is very efficient with blocktrees.

E Efficient Status Structures

Having improved the organization of payment information, let us now turn our attention to improving that of status information.

Recall that the official status at round r, S^r, consists of a list of tuples, specifying, for each current public key X, the amount owned by X, and possibly additional information: $S^r = \ldots, \left(X, a_X^{(r)}, \ldots\right), \ldots$ The amount of money owned by the keys in the system changes dynamically, and we need to keep track of it as efficiently as possible.

So far, as in Bitcoin (although its status information is quite different), the current status is not authenticated, but deduced from the authenticated history of payments. There are, however, significant advantages to authenticating each S^r as well, rather than only PAY^r. Indeed, this would make it easier and more secure, for a new user, or a user who has been off-line for a while, to catch up with the current system status. In particular, such a user X may have been chosen to be a round-r verifier, and thus needs to learn S^{r-1} in order to perform his duties in round r. Accordingly, X might ask and obtain the status S^{r-1} from another user, but then may worry about the accuracy of the information received. This worry could dissipate if S^{r-1} were authenticated. But: *who should authenticate the status at the end of a round?*

In Algorand, there is a natural choice: namely, (a given majority of) the verifiers of that round.[31] Indeed such verifiers, in order to perform their duties, presumably already know the previous status, and thus it is easy for them, after computing the current official payset, to compute and authenticate also the corresponding new status.

In sum, the round-r verifiers may conceptually operate as follows:

(a) Obtain the authenticated status S^{r-1};

(b) Compute, authenticate, and propagate PAY^r; and

(c) Compute, authenticate, and propagate S^r.

This way of operating presupposes that (at least a given majority of) the verifiers in TV^r already know, or can correctly obtain, TV^{r-1}. But obtaining TV^{r-1} is easier than correctly obtaining S^{r-1}. In particular, under all improved verifier-selection mechanisms of Section 6, each verifier

[30] Notice that this path is actually the concatenation of the authenticating path of $INFO_i$ in T_i and the "authenticating path" of R_i in T_z. Indeed, the notion of an authenticating path, defined for (the contents of) leafs, is readily extended to arbitrary nodes.

[31] Equivalently, the leader of that round may authenticate the resulting status together with his proposed payset for the round, and then the verifiers may verify both, and agree and certify the official payset and the resulting status together.

Y in TV^{r-1} provides a proof of his belonging to TV^{r-1}, and this proof may be easily requested, obtained, and verified.[32]

The above organization of a round may already be very beneficial. Indeed, it avoids that one must obtain the *entire payment history* in order to be sure about how much money each user currently owns. Yet, there is room for efficiency improvements.

Indeed, note that propagating an authenticated version of PAY^r can be done reasonably efficiently, because, although the number of users may be very large, the number of payments made in a single one-minute round, can be expected to be reasonably small. However, if there were —say— 100 million users in the system, and the status of each of them comprises —say— 100 bytes, then S^r would comprise 10 gigabytes.

Propagating such a large file is not trivial. Even authenticating it is not trivial. Indeed, computing $H(S^r)$ by hash-chaining, and then digitally signing the 32-byte result, would require computing one digital signature, which is trivial, but computing 100 million hashings (at about a microsecond per hash) would require 100 seconds. Only propagating the verifiers' digital signatures of S^r is trivial, because they are relatively few and short.

Accordingly, let us put forward a better *status structure*.

E.1 Efficient Status Structures via Blocktree Technology

Recall our use of blocktree, in Section D, in order to produce a new block B_r in a way that allowed both efficient block generation and efficient individual block provability.

At a high level, to generate a new block B_r at round r, we generated a Merkle tree T_r from the previous Merkle tree T_{r-1} and its associated short structural information $\mathcal{S}_{r-1}$, by

(a) storing the information of a new block, $INFO_r = v_r$, into empty leaf r of the underlying big binary tree T;

(b) using $\mathcal{S}_{r-1}$ in order to compute the contents of the nodes in the (shortest) path from leaf r to the (possibly new) root R_r of the new Merkle tree T_r; and

(c) computing the new structural information $\mathcal{S}_r$ to facilitate the construction of the next Merkle tree.

Note that T_r is the smallest Merkle tree that contains the first r leaves of T. In principle, to construct T_r one would have to compute, by hashing, the contents of all its nodes, in the typical bottom-up fashion, starting from its leaves. This would take a long time. It is having available the properly constructed structural information $\mathcal{S}_{r-1}$ that prevents us from having to handle all leaves and indeed all nodes of T_r.

In our new application, we plan a similar strategy. Namely,

Game Plan We plan to start with a big, underlying, and initially empty, binary tree T'; to progressively store status information in its leaves; to construct a sequence of Merkle trees, $T'_1, T'_2, \ldots$, where the leaves of T'_r collectively store the entire status information S^r; and to construct

[32] Recall that such a proof consists of a digital signature of Y, where Y was a key already existing a few rounds before $r-1$. But then, for most $Y \in TV^{r-1}$, Y is a much older key, and thus may be already known to a verifier X in TV^r, who at least learned some prior status. In any case, at least some information is necessary to verify any authenticated information. The point is to ensure that such necessary information is minimal, and easily known or easy to learn.

a matching sequence of structural information, $\mathcal{S}'_1, \mathcal{S}'_2, \ldots$, in order to facilitate the construction of future trees. $\triangle$

This is a simple enough plan. But: *How to choose $\mathcal{S}'_r$?*

Before answering this question, we should decide what status information is stored in a leaf. We could certainly store status information about multiple keys, but prefer, for conceptual simplicity, to store just the status information of a single key. Specifically, the ith leaf of T' stores only the tuple corresponding to ith key in the system, in order of appearance. (Recall that, in a round r, multiple keys may appear for the first time as payees in payset PAY^r. But since all sets in Algorand are ordered, including all paysets PAY^r, the ith key in order of appearance is well defined.) Having clarified this, we now construct the structural information $\mathcal{S}'_r$ in two stages. Better said, we let $\mathcal{S}'_i$ consist of two components, each corresponding to a difference between the Merkle tree T_r, of our previous block-structure application, and the Merkle tree T'_r, of our current status-structure application.

A First Difference, A First component of $\mathcal{S}'_r$ A first difference between the Merkle trees $T_1, T_2, \ldots$ and the Merkle trees $T'_1, T'_2, \ldots$ is the following. Each T_r had only one more (non-empty) leaf than T_{r-1}. However, T'_r may have many more leaves than T'_{r-1}.

More precisely, let n_r be the cardinality of PK^r, that is, the number of users at the start of round r, and q_r the cardinality of $PK^r \setminus PK^{r-1}$, that is, the number of new keys appearing as payees in PAY^r.

Then

- $n_{r+1} = n_r + q_r$;
- T'_r has n_{r+1} non-empty leaves, each storing the tuple corresponding to a key in PK^{r+1};
- the old leaves of T'_r (i.e., its first n_r leaves) continue to store the tuples corresponding to the old keys (i.e., the keys in PK^r);
- the new leaves of T'_r (i.e., its next q_r leaves) store the tuples corresponding to the new keys (i.e., the keys in $PK^r \setminus PK^{r-1}$); and
- the depth of T'_r is $\lceil \log n_{r+i} \rceil$.

Despite the fact that there may be $q_r > 1$ new leaves in T'_r, it is easy to see that the same type of structural information as in our block application suffices to enable the "efficient addition to T_{r-1}" of the new q_r leaves.

Recall that, in our previous application, $\mathcal{S}_r = (R_i, auth_r)$. Then, in our current application, let $\overline{\mathcal{S}_r}$ consist of R_r and the authenticating paths of all new leaves: that is, $\overline{\mathcal{S}_r} \triangleq \left(R_r, \left\{auth_r : v_r \in PK^r \setminus PK^{r-1}\right\}\right)$. Then,

The first component of $\mathcal{S}'_r$ is $\overline{\mathcal{S}_r}$.

The structural information $\overline{\mathcal{S}_r}$ suffices to "handle the inclusion in T'_{r+1}" of all new leaves, without having full knowledge of T'_i. Better said, $\overline{\mathcal{S}_i}$ suffices to efficiently construct a Merkle tree, T''_r, that (1) contains T'_{r-1} as a subtree, and (2) includes the new q_r leaves. (In fact, since the authenticating paths of the new leaves overlap, one may define $\overline{\mathcal{S}_r}$ more succinctly. We omit to do so since it is not going to yield significant improvements in efficiency.)

However, a second difference between our block-structure and status-structure applications prevents T''_r from being the Merkle tree T'_r we want.

The Second Difference, and the Second Conceptual Component of $\mathcal{S}'_r$ The second difference is that, while the old leaves of T_r have the same contents they had in T_{r-1}, the old leaves of T'_r may have different contents in T'_r than in T'_{r-1}. This is so because the payments of round r cause the amounts owned by some old keys to change. More precisely, letting t_r be the total number of payers and payees in PAY^r, then up to t_r old leaves of T'_r can have contents different from those they had in T_{r-1}. What makes it easy to "efficiently add to T'_r" the contents of the new leaves is that the new leaves are "neatly" positioned in T'_r. Namely they consecutively appear after the last old leaf. Thus, conceptually speaking, to add them in T'_r we need not know the entire tree T'_{r-1}, but only the information stored at its "frontier" (which, in a tree of logarithmic depth is not much information at all).

By contrast, in the official payset PAY^{r-1}, the old payer and payee keys may appear anywhere in the old leaves of T'_r. But this is far from implying that we need the whole T_{r-1} in order to construct T'_r correctly. Indeed whenever, due to a payment in PAY^{r-1}, the content of an old leaf i changes, only the contents of the path $\mathcal{P}_i$, that is, the (shortest) path from leaf i to the root, need to be updated, and this path is short (indeed logarithmic in n_r). Moreover, the number of payments in PAY^{r-1}, and more generally the number of payments in any one-minute round, should be small relative to the total number of public keys in the system. This leads to a hypothetical second component of $\mathcal{S}'_r$ as follows.

Let X be an old key whose content, due to the payments in PAY^{r-1}, has changed. That is, $\left(X, a_X^{(r-1)}, \ldots\right) \neq \left(X, a_X^{(r)}, \ldots\right)$. If X is the ith key, then, to reconstruct the contents of the nodes of $\mathcal{P}_i$ in T'_r, it suffices to have the authenticating path of leaf i in the Merkle tree T'_{r-1}. Identifying each key with its number of appearance, we may denote this authenticating path by $auth_{r-1}(X)$, and refer to leaf i as leaf X.

As we said, such an authenticating path is very short. More precisely, it consists of $\lceil \log n_r \rceil$ values of 32-bytes each. Since $\log 10^9 < 30$, in a system with a billion users $auth_{r-1}(X)$ is at most one kilo-byte of information. Moreover, given $auth_{r-1}(X)$ and the new content $\left(X, a_X^{(r)}, \ldots\right)$ of leaf X, computing the contents of all nodes in the path from leaf X to the root of T'_r only requires, again in a system with one billion users, at most 30 hashings, which is a trivial amount of computation. Thus, conceptually, the second component of $\mathcal{S}'_{r-1}$ we need to finish the construction of T'_r consists of all such authenticating paths:

$$\textit{The conceptual second component of } \mathcal{S}'_{r-1} \textit{ is } \widetilde{\mathcal{S}'_{r-1}} = \{auth_{r-1}(X) : X \textit{ old }\}.$$

The problem, of course, is that no matter how helpful the previous-round verifiers want to be, they cannot provide $\widetilde{\mathcal{S}'_{r-1}}$, because they cannot predict what payments the users will make in the next round. Thus: *how can the round-r verifiers obtain $auth_{r-1}(X)$?*

A New Requirement In a payment system, each user X has a personal interest in keeping —and in fact *safekeeping*— a proof of the current amount he owns. (In addition, other users may ask X for such a proof. For instance, if X wishes to make a payment P to a payee Y in a round r, then Y may demand to see a proof that X indeed has enough money to make at least that payment.)

More generally, a user X may wish to have a proof of his own full status, S_X^r, that is, at a round r, of the tuple $S_X^r = \left(X, a_X^{(r)}, \ldots\right)$.

In the envisaged status structure, a proof of S_X^{r-1}, the status of X at the start of round r, precisely consists of the authenticating path $auth_{r-1}(X)$ in tree T'_{r-1}.

Thus, assuming for a moment that each user X can easily obtain such a proof, we may require that, for a round-r payment P with payer X and payee Y, also the proofs $auth_{r-1}(X)$ and $auth_{r-1}(Y)$ be presented. For simplicity, let us assume that they are actually explicitly included in P itself (e.g., as part of P's information field I) and that they must be properly verified for P to be valid. To emphasize that the payments are now so enriched, we shall denote the official payset of a round r by $\overline{PAY^r}$. (Let us stress again that $\overline{PAY^r}$ is essentially comparable in length to PAY^r.)

Then, the second component of $\mathcal{S}'_r$ consists of the authenticating paths, *in the Merkle tree* T'_r*!*, for each $X \in OldKeys\,(PAY^r)$, that is for each key X whose content changes due to the payments in PAY^r:

$$\widetilde{\mathcal{S}'_r} = \{auth_r(X) : X \in OldKeys(PAY^r)\} \ .$$

We now make two claims:

(a) The verifiers of a round r, given $\overline{PAY^{r-1}}$, $\overline{\mathcal{S}_{r-1}}$ and $\widetilde{\mathcal{S}_{r-1}}$, can determine $\overline{PAY^r}$, $\overline{\mathcal{S}_r}$ and $\widetilde{\mathcal{S}_r}$.

(b) Every user Z, whether or not being a payer or a payee in round r, can, from $\overline{PAY^r}$, $\overline{\mathcal{S}_r}$, $\widetilde{\mathcal{S}_r}$, and $auth_{r-1}(Z)$, efficiently compute $auth_r(Z)$, that is a proof of his status S_X^r.

(In fact it is because of this ability of the users that the round-r verifiers can receive the enriched official payset $\overline{PAY^{r-1}}$ instead of the old PAY^{r-1}.)

Let us first informally prove the first claim.

First of all, once the users propagate the payments of round r, the round-r verifiers, and the round leader construct $\overline{PAY^r}$ as usual (e.g., as in the preferred version of Algorand).

Second, the round-r verifiers can construct (and then authenticate) $\overline{\mathcal{S}_r}$ from $\overline{\mathcal{S}_{r-1}}$ as explained above.

Third, the round-r verifiers retrieve, from the enriched payments in $\overline{PAY^r}$, the authenticating paths $auth_{r-1}(X)\ T'_{r-1}$ for any $X \in OldKeys(PAY^r)$, and, from these authenticating paths and $\overline{\mathcal{S}_r}$, they finally produce the new authenticating paths in T'_r that constitute $\widetilde{\mathcal{S}_r}$.

Let us now informally argue the second claim.

A user Z can compute $auth_r(Z)$ by the following three steps.

1. For each old key $X \in OLdKeys(PAY^r)$, Z uses the new content of leaf X and $auth_r(X)$ to compute both $path_r(X)$ and $content_r(X)$: respectively (a) the sequence of nodes in the (shortest) path from leaf X to the root of T'_r (in that order) and (b) the sequence of the contents of the nodes in $path_r(Z)$ (in the same order).

2. If one node in $path_r(X)$ coincides with the sibling of a node in the path, in T'_{r-1}, from leaf Z to the root of T'_{r-1}, then Z replaces in $auth_{r-1}(Z)$ the content of that node with the content of the same node in $content_r(X)$.

   Let $auth_{r-1}(Z)'$ be the resulting sequence after all such replacements.

3. From the structural information $\overline{\mathcal{S}_r}$, Z computes the authenticating path of the root of T'_{r-1} in the new tree T'_r, $auth_r(R_{r-1})$, and then appends this authenticating path to $auth_{r-1}(Z)'$.

   It is easy to see that the resulting sequence is Z's new authenticating path $auth_r(Z)$ in T'_r.

Summarizing In sum, we have organized the status information in a Merkle-tree fashion, so that (1) each user has a short proof of his own status at each round; (2) each payment at a round r is enriched so as to include a proof of round-$(r-1)$ status of the payers and the payee; (3) it is possible for the verifiers of a round r to process all payments and produce the (enriched) official payset, using only a very small amount of information from the previous round (rather than the entire status of the previous round); and (4) it is possible for every user, possessing a proof of his own status in the previous round, to efficiently compute a proof of his status in the current round, whether he has made any payments or not.

Notice that this organization puts some burden on the users who want to make a payment, but (a) putting one's financial status in order when making a payment may not be an unreasonable requirement; (b) a user is *not* required to update the entire status information, but only his own small piece of it; (c) if a user does not keep updating his own status, then he *does not* need to recompute it, round-by-round, from the last time in which he did: all he has to do is to request from someone else just his own current and *short* proof of his current status, a proof whose correctness he can *verify* based on the same short structural information used by the verifiers; and (d) the verifiers need only obtain a short amount of information, if they have been off-line for a long time, in order to be fully prepared to perform their duties.

Finally, (f) if so desired, as in our block application, one can efficiently prove the amount owned by the users at a given round.

One Important Protection As we have said, the status of a player X at a round r, $S_X^r = \left(X, a_X^{(r)}, \ldots\right)$, may possess several fields in addition to "his own name" and "the amount he owns". Let us suggest that one of these fields, say the third, be a self-certified declaration of how much X owns at a round $q \leq r$: for example, $SIG_X\left(X, q, a_X^{(q)}\right)$. Thus, the round-$r$ status of X is of the form

$$S_X^r = \left(X, a_X^{(r)}, SIG_X\left(X, q, a_X^{(q)}\right)\ldots\right).$$

When X is the payer or the payee of a payment P in a round r, P should include, as already said, the proof of his status at round $r-1$, $auth_{r-1}(X)$, which of course authenticates also the third field. But for P to be a valid round-r payment, not only should $auth_X^{(r-1)}$ be correct, but the third field should actually be "up-to-date": that is, it must indicate the last round, $SIG_X\left(X, r-1, a_X^{(r-1)}\right)$, and must correctly report the amount X actually ownwed at the end of round $r-1$.

There are several clear advantages in so enriching the status information.

Acknowledgements

I would like to first acknowledge Sergey Gorbunov, my coauthor of the cited Democoin system, which provided the foundation of Algorand: relying on a cryptographically chosen and ever-changing committee of verifiers.

My 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 exposition of an earlier version of this paper. (Unfortunately for the reader, I have further revised it, so I am solely to blame for its current deficiencies.)

Thanks to 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, and for informing me that he and David Karger had already thought of a similar way of structuring payments in a public ledger.

Thanks to Jing Chen, for several helpful discussions and her usual patience, kindness, and unfailing wisdom.

Thanks to my colleagues Arvind, Hari Balakrishnan, Frans Kaashoek, Robert Morris, Martin Rinard, and Nikolai Zeldovich for trying to explain to a naive theoretician (to put it mercifully) how modern Internet communication protocols work.

Thanks also to David Lazar and Yossi Gilad for their interest in this technology and for running some initial experimental tests.

References

[1] *Bitcoin Block Chain Info*, https://blockchain.info, Feb 2015.

[2] 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.

[3] Wikiquote.org. *William F. Buckley Jr.*, https://en.wikiquote.org/wiki/William_F._Buckley,_Jr. As of 5 June 2016.

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

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

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

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

[8] 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.

[9] Wikipedia. *Numbers Game* https://en.wikipedia.org/wiki/Numbers_game As of 5 June 2016.

[10] D. L. Chaum, *Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms*, Commun. ACM, Volume 24, Number 2, Pages 84–90, 1981.

[11] *Bitcoin Computation Waste*, http://gizmodo.com/the-worlds-most-powerful-computer-network-is-being-was-504503726, 2013.

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

[13] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*, 2008.

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

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

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

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

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

[19] 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.

[20] M. Rabin. *Randomized Byzantine generals.* Proc. 24th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1983. pp. 403-409.

[21] 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.

[22] P. Feldman and S. Micali. *18. An Optimal Probabilistic Algorithm for Synchronous Byzantine Agreement.* (Preliminary version in STOC 88.) SIAM J. on Computing, 1997

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

[24] *Litecoin*, https://litecoin.org/, 2011.

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

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

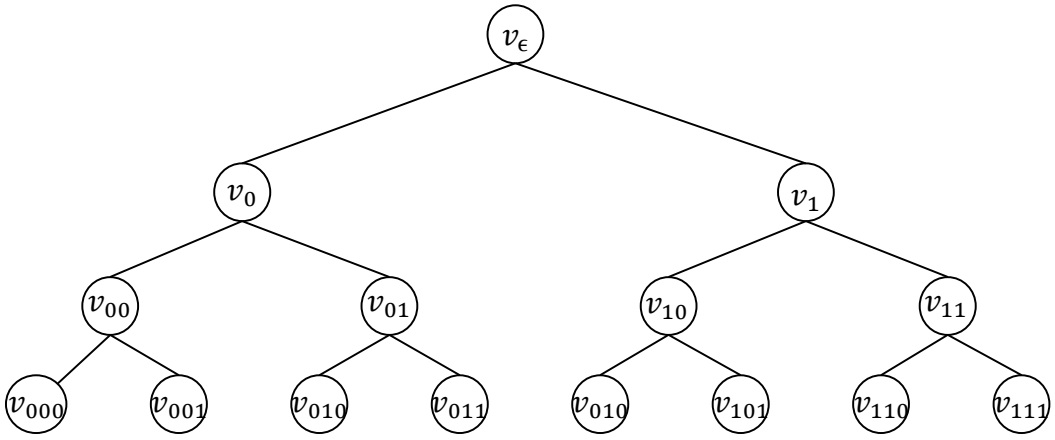

Fig. 1.A A Full Merkle Tree of Depth 3

$v_s = H(v_{s0}, v_{s1})$ for all binary string $s \in \{0,1\}^{\leq 3}$

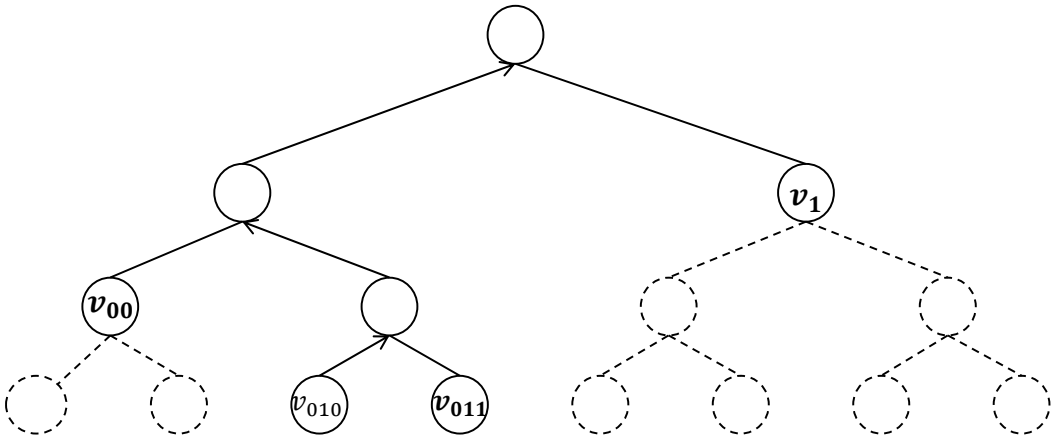

Fig. 1.B The Authenticating Path of Value v_{010}

The path from v_{010} to the root can be found by following the arrows.
The contents of the siblings of the nodes in the path are shown in bold font.
The authenticating path of v_{010} is the (bottom-up) sequence of the siblings' contents: v_{011}, v_{00}, v_1.
The contents of all nodes in the path can be computed from v_{010} and its authenticating path via H.
All other nodes are ignored.

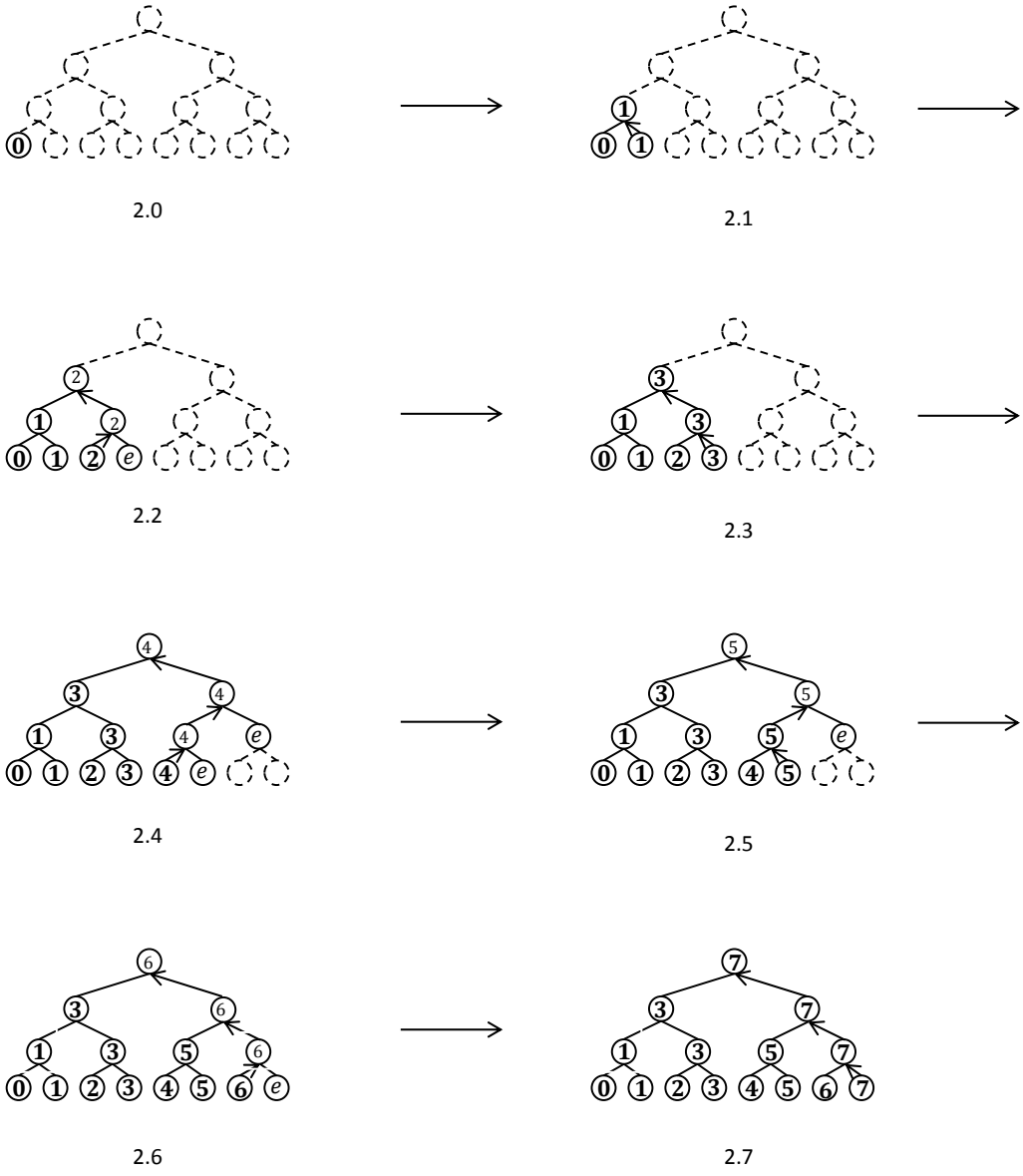

Fig. 2 The First 8 Merkle trees constructed within a full binary tree of depth 3.

In Fig 2.i, nodes marked by an integer belong to Merkle Tree T_i.
Contents of nodes marked by i (respectively, by $\boldsymbol{i}$) are temporary (respectively, permanent).

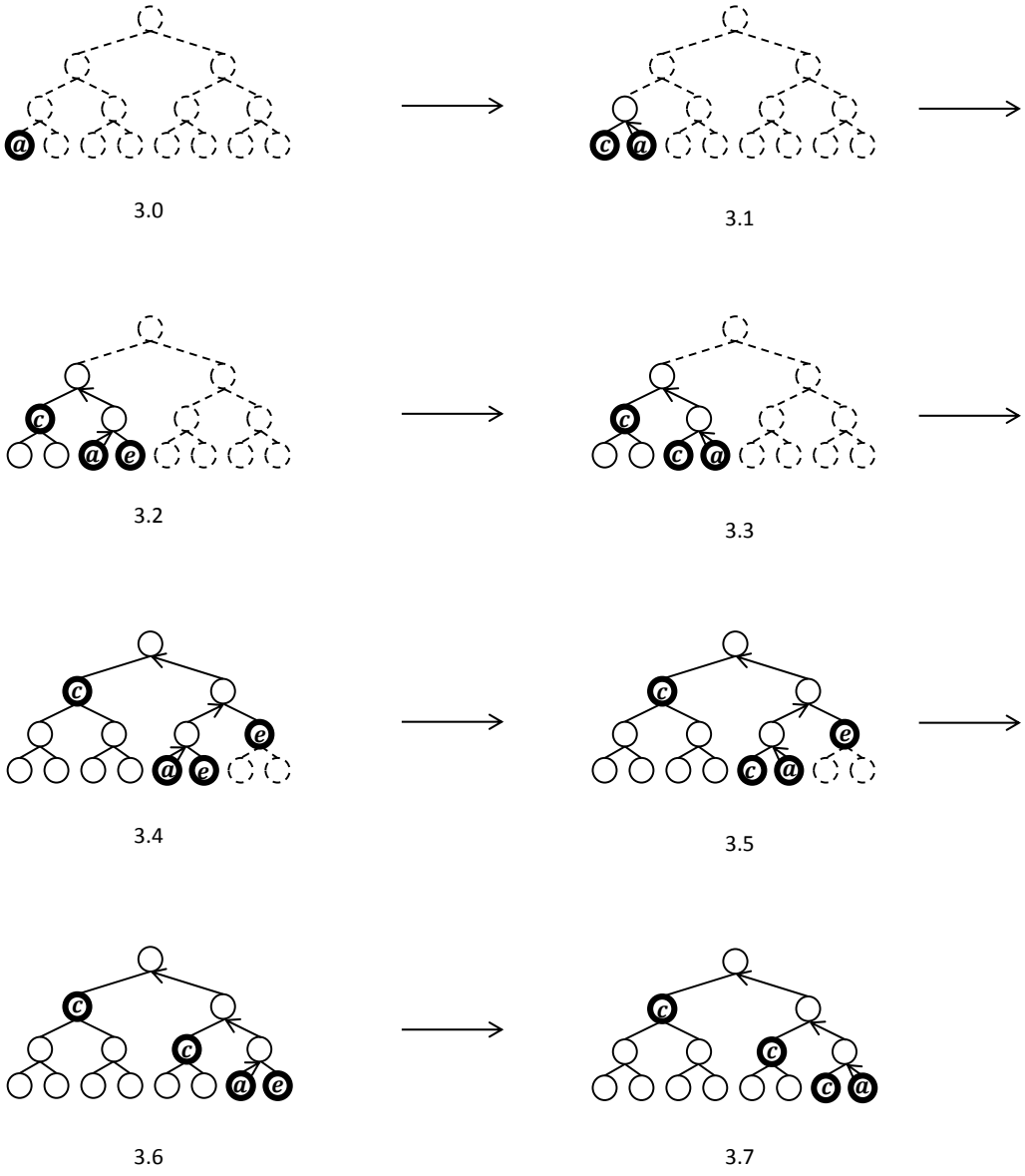

Fig. 3 The Items sufficient to construct the structural information of the first 8 blocks in a blocktree.

In Fig 3.i, the contents of the boldly circled nodes suffice to construct the structural information of block B_i. For instance, the node marked c in 2.2 is the root R_1 of tree T_1, which is explicitly part of block B_1.