Modeling and Verification of the Algorand Consensus Protocol

Musab A. Alturki
Runtime Verification, Inc.
King Fahd University of Petroleum and Minerals
musab@kfupm.edu.sa

Brandon Moore
Runtime Verification, Inc.
brandon.moore@runtimeverification.com

Karl Palmskog
The University of Texas at Austin
palmskog@utexas.edu

Lucas Peña
Runtime Verification, Inc.
University of Illinois at Urbana-Champaign
lpena7@illinois.edu

Grigore Roşu
Runtime Verification, Inc.
University of Illinois at Urbana-Champaign
grosu@illinois.edu

Abstract

The Algorand consensus protocol is at the core of the Algorand platform for secure and decentralized digital currencies and transactions. In this report, we describe our effort to model and formally verify the Algorand consensus protocol in the Coq proof assistant. We give an overview of the protocol, outline our model of the protocol as a state transition system, and describe how its properties are formalized and proved. A key contribution of this work is the elucidation of the assumptions under which the main safety property of the protocol holds. The Coq source files are available at: https://github.com/runtimeverification/algorand-verification

1 Introduction

Algorand is a platform for secure and decentralized digital currencies and transactions [13]. At the core of the Algorand platform is the Algorand consensus protocol [2, 3], a pure Proof-of-Stake (PoS) protocol that provides efficient, secure, and scalable operation while remaining decentralized. The basic idea in pure PoS is to make the security of the system dependent only on how honest the majority of its asset owners are, without having to rely on any specific subset of participants or having to lock up assets and penalize users.

Distributed consensus protocols are complex, and subject to general limitations even in non-adversarial settings [5]. They typically involve interaction between many independent and potentially untrusted nodes in a large network where asynchronous and failure-prone communication is the norm rather than the exception. Algorand, in particular, specifies not only the node-local behavior, but also deals with message propagation and delivery, to be able to address two types of attacks: (1) attacks that corrupt participating nodes so that they no longer follow the protocol, e.g., to produce fake transactions or to cast votes for the wrong transactions, and (2) attacks on the underlying network in which the system is deployed, for instance by intercepting, manipulating, and delaying messages. When an attacker gains full control of message delivery in a network, the network is said to be partitioned. Unlike Algorand, most existing consensus protocols do not consider what happens when a network is partitioned or after the network recovers from a partition.

Consensus systems are now the backbone of large cryptocurrencies worth billions of US dollars. This means that vulnerabilities can have catastrophic consequences. Consequently, ensuring correctness and resilience against malicious behavior while designing a consensus protocol is an important and challenging task. Having strong guarantees of correctness and developing a thorough understanding of protocol assumptions can significantly reduce the risk of catastrophic events.

In this report, we describe our effort to model and verify the Algorand consensus protocol using deductive verification. In this approach, systems are modeled and specified inside expressive formal logical systems, and verified in a similar way to how mathematicians prove theorems - in principle by elaborating proofs of statements step-by-step. In the spectrum of formal verification techniques, deductive verification provides the strongest guarantees, and thus the highest degree of trustworthiness, but at the expense of being the most demanding. Specifically, we use Coq [1], a proof assistant developed for more than 30 years and used in several large-scale verification projects [11, 19].

We developed a model of Algorand consensus in Coq in the form of a *transition system*, over which we stated and ultimately proved the *asynchronous safety* property: no two honest nodes certify two different blocks, even when the adversary has complete control of message delivery in the network. We also precisely pin down the assumptions under which the safety theorem holds. Beyond the safety theorem,

Project Report, June 18, 2019, Runtime Verification, Inc.
2019.

we intend for this formalization to lay the foundation for further modeling and verification efforts related to the Algorand consensus protocol.

2 Background

This section generally introduces blockchain and consensus protocols, gives an overview of Algorand, and provides the pertinent Coq background.

2.1 Blockchains and Consensus Protocols

A blockchain is an ordered sequence of cryptographically linked blocks of records, in much the same way as links in a chain are firmly latched in sequence. A blockchain provides a persistent, tamper-proof and globally accessible ledger of transactions. It is built using well-established and publicly known cryptographic tools, including most importantly one-way hash functions. Anyone can verify the validity of its transactions, and no one can tamper with a transaction or claim a transaction exists that does not appear in the chain.

This makes blockchains particularly well suited for building self-governing and autonomous distributed systems. Indeed, a blockchain is a key component for allowing a collection of nodes in a communication network (who do not necessarily trust each other) to work collectively and make decentralized decisions. Nevertheless, the nodes will need more than just a blockchain to achieve proper decentralized operation. Specifically, they need a mechanism for identifying and agreeing on the next block of transactions to be appended to the chain, a mechanism known as a consensus protocol.

A consensus protocol typically proceeds in rounds, where the objective of a round is to produce the next block and record it in the chain. As such, a consensus protocol includes both: (1) a mechanism for decentralized selection of one or more block proposers or producers of a round in the protocol, and (2) a mechanism for achieving decentralized consensus on a single block to be appended to the chain.

In an idealistic (and rather unrealistic) setup, in which communication networks are perfectly reliable with zero-delay message delivery, and in which all participating nodes run error-free code and behave honestly, achieving consensus on a single block in each round is a trivial task. All nodes would instantaneously see all proposed blocks and their trusted opinions about them. In reality, however, the situation is very different. Distributed systems utilize global Internet (IP-based) communication networks, which are inherently unreliable and where significant message transmission delays are not uncommon. Furthermore, nodes may deviate from the protocol, either intentionally (when compromised by malicious users) or unintentionally (due to internal errors). These complications can hinder the consensus process, resulting potentially in the system being unusable or in losing large amounts of assets maintained by the system.

Therefore, consensus protocols need to maintain a consistent global view of the system while relying only on the local knowledge at the level of its individual (honest) nodes. Moreover, these protocols must ensure continuous and fair operation of the system in the presence of both benign node or network failures and maliciously behaving nodes. Consequently, consensus protocols are inherently complex, consisting of multiple asynchronous steps and utilizing a wealth of cryptographic and randomization schemes, and possibly some economic incentive structures, whose goal is to reward compliant behaviors and penalize deviations from the protocol.

A class of consensus protocols, pioneered by Bitcoin [14] and referred to as Proof-of-Work (PoW), achieves consensus primarily through a process called mining, in which all participating nodes in the network compete on solving complex cryptographic puzzles to produce and record blocks. Although it provided workable solutions to the decentralized consensus challenge, PoW is now widely known to suffer from efficiency, scalability, and security problems. The mining process is inherently computationally expensive, slow, and wasteful, resulting in wasting significant amounts of energy while imposing high transaction fees. Furthermore, PoW tends in practice to result in centralized systems, in which the bulk of the mining power (and hence the power of deciding the fate of the blockchain) falls in the hands of a small subset of users: those who can afford to invest in very powerful computing resources, or who can simply join forces in mining pools. Centralization defeats the purpose of the protocol and poses a major security concern: those who control the majority of the mining power need to be trusted, and even if they are, they make the system a much easier target for attacks.

An alternative mechanism that promises to alleviate these problems is Proof-of-Stake (PoS), in which the burden of producing blocks and selecting the next block to augment the blockchain is placed on a suitably selected subset of nodes (typically much smaller than the entire set of participating nodes), called a committee. In some variations of PoS, called delegated PoS, the committee is trusted and is usually fixed. In other variations, generally referred to as bonded PoS, participation in a committee requires staking some of the blockchain's underlying cryptographic assets (such as cryptocurrency or tokens), or locking up these assets for an extended period of time so that they cannot be expended or moved. The stake of a committee member (relative to the total stake in the system) determines the member's voting power when deciding the next block for a given round in the protocol. Furthermore, a node's stake can also be used as collateral to be (fully or partially) reclaimed by the system if a node is found to misbehave.

However, many important questions arise when designing a PoS protocol. How is the committee selected? One would

want to have a selection process that is fair and representative, and hard to manipulate by a malicious user. Moreover, a committee that is fixed for a prolonged period of time can be easily attacked and presents a potential single point-of-failure for the system. But how often should the committee be changed? What is a suitable size of the committee in relation to the total population of nodes in the network? How should the voting process be designed? These and other design choices can significantly affect the efficiency, scalability and security of a PoS blockchain system.

2.2 Overview of Algorand

We highlight below some of Algorand's unique features. More detailed descriptions of these features and others are available elsewhere [2, 3].

Algorand almost never forks. Forking happens when consensus on a single block for a round is not reached and multiple different blocks are added for that round. This is a notorious problem in PoW, but is also possible in PoS systems. Having different subsets of nodes decide on appending different blocks to the chain means that transactions appearing in these conflicting blocks are not finalized since only one of these blocks will eventually belong to the canonical chain, the chain that is deemed most accepted by the nodes (there are different methods for deciding the canonical chain, e.g. the longest chain in Bitcoin). Algorand avoids this problem by design: at most one block can receive the majority of votes in a round.

Algorand is very efficient. In Bitcoin and other PoW protocols, the rate at which blocks are produced (and hence the rate at which transactions are processed in the blockchain) is determined by the complexity of the cryptographic puzzle to be solved by the miners: the more complex the problem is, the longer it takes to produce a block. Although simplifying these puzzles would mean faster block production, the effective transaction processing rate would likely be adversely impacted as the forking rate will also increase. More miners will now compete to have their blocks in the chain and many more transactions will now belong to chains that end up being abandoned (they will have to be re-processed and attempted again). Being a PoS system, Algorand does not have this problem, and is in fact designed to produce a block every second. This high block production rate when combined with the fact that the chain forks only with negligible probability means that transactions appear very quickly in the chain, and once they do, they can immediately be considered final. This has the potential of allowing extremely high levels of scalability.

Algorand provides security against two types of attacks. First, attacks on the protocol, which involve corrupting participating nodes so that they no longer follow the protocol, e.g., by producing blocks with fake transactions or by casting votes for the wrong block or casting multiple conflicting votes. Second, attacks on the underlying network on which the system is deployed, for instance by intercepting, manipulating and delaying messages. When an attacker gains full control of message delivery in a network, the network is said to be partitioned. Most existing consensus protocols do not consider what happens when a network is partitioned or after the network recovers from a partition. An attacker's goal is generally to manipulate the chain to reverse transactions or rewrite history (e.g. double-spend attacks), or to attempt to break consensus, leaving the system in an inconsistent state, or even cause the system to halt, where the chain can no longer grow.

The key to Algorand's security in its pure PoS protocol is maintaining decentralization at every step of the protocol. This is generally achieved through several techniques relating to how a committee, which is a group of nodes selected to make a decision on behalf of all nodes in the network, is managed. We highlight below key aspects of committee management.

Committee membership. Membership in a committee is decided through a unique process called *cryptographic self-selection*, which is run individually by each node. Essentially, a node locally runs a cryptographically fair and irrefutable lottery whose outcome (which can be verified by everyone else) decides membership in the committee.

Committee formation. Committees are not fixed and change very frequently throughout execution of the protocol. In fact, every step of execution may have its own committee. Furthermore, committees can be of different sizes depending on the type of task the committee is entrusted to perform. For instance, the committee responsible for producing a block is typically much smaller than the committee that votes on blocks.

Attackers and committees. Cryptographic self-selection is performed secretively by each node in isolation of all other nodes. Only the node knows whether it is part of the current-step committee, until it announces its membership with its vote or block to the network. An attacker has no way of knowing beforehand whether a node belongs to a committee. Once the node announces its membership, it will already be too late for an attacker to corrupt the node and send a different message. The node's original messages are already out and being propagated through the network (note also these messages are digitally signed, so they cannot be tampered with without the receiver knowing).

Ephemeral keys. Another distinguishing feature of Algorand is the use of ephemeral keys, which are temporary, single-use encryption-decryption keys, for signing vote and block messages. Once a node uses an ephemeral key to sign a message, the node immediately destroys it, so that if the node gets compromised later on by an attacker, the attacker won't be able to claim that the node sent a different message for a previous step in the protocol. This effectively prevents

an attacker from going back in history and forging different messages from what the nodes originally sent.

By performing cryptographic self-selection at every step of the protocol and using ephemeral keys for signing block and vote messages, Algorand decouples the execution of a protocol step from the identities of nodes participating in that step, a property referred to as *player replaceability*. Indeed, the execution of a step does not depend on the state of a node, and a node is never obliged to participate in a minimum number of steps in the protocol.

The Algorand consensus protocol proceeds in rounds and any node can participate in the protocol (i.e. Algorand is permissionless). A new block is generated at each round. A round is divided into periods, and a period is further divided into steps. A potential block is proposed in each period of a round, and the round ultimately ends when the block is finalized.

The four kinds of steps in each period are the proposing step, filtering step, certifying step, and finally the finishing step(s). Present in each of these steps is Algorand's cryptographic self-selection process highlighted above. That is, a verifiable random function is used to privately and securely select committee members at each of these steps. The random, verifiable, self-selecting nature of this process is key to the correctness of the Algorand protocol.

Proposing step. Committee members in this first step (chosen by cryptographic self-selection) are known as potential leaders. These members propose a block and propagate it to other members of the protocol.

Filtering step. Here, committee members identify their leader, also using cryptographic self-selection. Committee members evaluate their current state, and potentially soft-vote for the proposed block.

Certifying step. In the certifying step, committee members simply evaluate if there are enough soft-votes for the potential block. If so, each committee member submits a cert-vote for that block. If there are enough certvotes for a block, the block is ultimately approved or certified, and users move to the next round.

Finishing steps. The last stage of a period, the finishing steps allow users to move to a new period without certifying a block. Here, committee members evaluate the proposed block, as well as votes received from other members, including votes in the last period. This allows users to potentially propagate a next-vote message. Sufficient next-votes result in the period increasing without a block being certified.

A node may proceed to the next period of a round when it observes a quorum of next-votes for a block (or for a special bottom value indicating that no potential block has enough votes yet). Once a node observes that a block gets a certificate, which is a large-enough quorum of cert-votes, the node certifies the block and moves on to the next round.

2.3 Deductive Verification and Proof Assistants

One emerging approach to the challenge of designing and implementing correct and robust consensus protocols is to apply formal methods during design and development. More specifically, in this approach, engineers formally specify both a protocol's design and the properties that the design is required to meet. Then, they mathematically verify that the design meets the requirements, or, if it does not, determine how the design may violate the requirements. In the latter case, they can use their findings to revise the design. When a design is successfully verified, engineers also precisely pin down the assumptions under which the design satisfies its requirements, which are important to consider when deploying an implementation of a design and for systems built on top of an implementation.

The benefits of applying formal methods early in the design process of complex systems are well documented [6, 20]. Most directly, formal methods can uncover fundamental errors that would otherwise go undetected, and which are costly to correct in a later phase. Moreover, obtaining formal guarantees about a protocol's design significantly increases the confidence in not just the implementation of the protocol, but also in the systems built on top of it, and ultimately increases trust in whole platforms among developers and users, facilitating wider adoption and support.

Researchers have developed a large spectrum of techniques and tools for applying formal methods to model, specify, and verify general software systems and distributed protocols. The techniques vary across dimensions such as expressive power, automation, and tool support. Perhaps most importantly, they differ in the mathematical guarantees they provide and therefore in trustworthiness. Generally, to obtain stronger guarantees, more effort is required by formal verification engineers during both modeling and verification, and less extensive automation is available.

The strongest guarantees, and thus the highest degree of trustworthiness, are obtained through deductive verification and theorem proving. In deductive verification, systems are modeled and specified inside expressive formal logical systems, and verified in a similar way to how mathematicians prove theorems - in principle by elaborating proofs of statements step-by-step. Among tools for deductive verification, *proof assistants* [8, 10] offer the smallest trusted computing base necessary to trust verified statements, realized in modestly-sized trusted checkers that only assume a small set of axioms accepted by almost all mathematicians. Although they provide such strong guarantees, proof assistants can offer extensive automation and user support. Nevertheless, the process of figuring out what properties are relevant, how these properties are specified and building up

their proofs normally requires a great deal of human ingenuity and experience. However, once everything is specified, proofs developed in proof assistants can be machine-checked, and persisted to serve as independently verifiable evidence that the properties hold for the given system [16].

To model and formally verify the Algorand consensus protocol, we use deductive verification and theorem proving, as facilitated by the Coq proof assistant [1].

2.4 Coq and the Mathematical Components Library

Coq is a proof assistant based on higher-order type theory, and can be viewed as consisting of, on the one hand, a small and powerful purely functional programming language, and on the other hand, a system for specifying and proving properties. A Coq user writes functions and data and then interactively constructs the proof of a theorem by trying different proof tactics that transform the state of the in-progress proof. Coq only accepts the theorem after its checker has been run on the purported proof. The Mathematical Components (MathComp) project [12] provides a set of Coq libraries with standard mathematical data structures such as sequences, finite sets, and multisets [4], packaged to facilitate extensions and formal proofs [7]. Proofs in MathComp are written in the SSReflect proof language [9], which extends Coq's proof tactic language.

3 Formal Model and Verification Approach

We model the Algorand consensus protocol in Coq as an inductive binary relation on global protocol states, and then express key properties as *invariants* that hold for all protocol states that are (transitively) reachable, according to the relation, from some initial state. Ultimately, we state and prove the *asynchronous safety* property of our model in this way.

Our verification approach is similar to that used for the Raft consensus protocol [19] and the Toychain blockchain formalization [15], but is different from, for example, the approach based on reasoning on Lamport's happens-before relation used to verify the PBFT protocol [17]. We also elide distributed separation logic, which has been used recently for consensus protocol verification in Coq [18].

As outlined below, our protocol model, properties, and proofs rely heavily data structures and results from the Mathematical Components project [12], and in particular its libraries for finite sets and finite maps [4]. In total, our Coq development contains around 2000 lines of Gallina code for the specification of the model and its properties, and around 4000 lines of proof scripts that build the formal proofs.

3.1 Assumptions

Most formalizations of programs and protocols are parameterized in specific ways, to be valid across a wide range of configuration choices and scenarios that may occur in practice. Our Algorand model is parameterized on a finite set of *user identifiers* (e.g., names of network nodes), and on a finite set of *values* used in message payloads, abstractly representing blocks and block hashes.

```
Parameters (UserId : finType) (Value : finType).
```

Values are assumed to be checkable for validity. We also consider an arbitrary, but totally ordered, datatype of credentials owned by users, and assume there is some way, given a credential, to say whether its owner is a committee member. Finally, we parameterize on delays and Algorand-specific configuration parameters such as number of votes required to move between periods. For example, these real-number parameters represent, respectively, the non-block message delivery delay, the block message delivery day, and the recovery time:

```
Parameters (lambda : R) (big_lambda : R) (L : R).
```

Besides these and other parameters (16 in total), we also express assumptions, most related to our parameters, in the form of axioms. We use four kinds of axioms: (1) standard axioms from classical logic, (2) axioms about credentials, (3) axioms on limits of message delays, and (4) axioms on the composition of groups (quorums) of users satisfying certain conditions.

We use the first kind of axiom (three in total) to facilitate simpler use of and reasoning about real numbers. One example of the second kind of axiom (out of two in total), expresses the uniqueness of credentials across different users:

```
Axiom credentials_different :
 ∀ (u u' : UserId) (r r' p p' s s' : nat), u ≠ u' →
 credential u r p s ≠ credential u' r' p' s'.
```

In other words, any two distinct user identifiers `u` and `u'` will have distinct credentials no matter what the round, period and step values are. The two axioms of the third kind specify how the small message delivery delay `lambda`, block message delivery delay `big_lambda`, and the recovery time period `L` are all related:

```
Axiom delays_order : (3 * lambda <= big_lambda < L)%R.
Axiom delays_positive : (lambda > 0)%R.
```

As one example of the fourth kind of axiom, the assumption that "any two large-enough cert-voting committees for a given round-period-step triple must share at least one honest user", which captures the honest supermajority assumption in Algorand for the cert-voting case, is specified as:

```
Axiom quorums_c_honest_overlap :
 quorum_honest_overlap_statement tau_c.
```

where `quorum_honest_overlap_statement` is a definition parameterized by the committee size threshold value `tau`:

```
Definition quorum_honest_overlap_statement (tau : nat) :=
 ∀ (trace : seq GState) (r p s : nat) (q1 q2:{fset UserId}),
  q1 `<=` committee r p s → #|` q1 | >= tau →
  q2 `<=` committee r p s → #|` q2 | >= tau →
  ∃ (honest_voter : UserId),
   honest_voter ∈ q1 ∧ honest_voter ∈ q2 ∧
```

```
  honest_during_step (r,p,s) honest_voter trace.
```

In other words, if for any two quorums of users, each of size at least `tau`, and who are all committee members for the given `(r,p,s)` triple, there is an honest user for the step `(r,p,s)` who belongs to both quorums. In total, there are six axioms of this kind.

Our axioms either capture statements accepted by mathematicians (1 above) or assumptions made by Algorand designers on the runtime environment (2-4 above).

3.2 Local and Global State

We model local (user) state as an inductive datatype that holds information such as the current round and period, as well as blocks and votes seen during the round.

```
Record UState := mkUState {
  corrupt : bool;
  round : nat;
  period : nat;
  step : nat;
  timer : R;
  deadline : R;
  p_start : R;
  proposals : nat → nat → seq PropRecord;
  stv : nat → option Value;
  blocks : nat → seq Value;
  softvotes : nat → nat → seq Vote;
  certvotes : nat → nat → seq Vote;
  nextvotes_open: nat → nat → nat → seq UserId;
  nextvotes_val : nat → nat → nat → seq Vote
}.
```

We model the global state as an inductive datatype that holds the current time, whether the network is partitioned, current state for all users, messages in transit, and received messages. In particular, the state of users in the global state is represented as a finite map from user identifier to state, and messages in transit as a finite map from user identifier to multisets of pairs of message deadlines and messages.

```
Record GState := mkGState {
  now : R;
  network_partition : bool;
  users : {fmap UserId → UState};
  msg_in_transit : {fmap UserId → {mset R * Msg}};
  msg_history : {mset Msg}
}.
```

3.3 The Global Transition Relation

The global state transition relation is defined over the `GState` type, and denoted `g ~~> g'` in Coq when it holds for `g` and `g'`. We distinguish three kinds of transitions: (a) node-internal transitions, (b) node message delivery transitions, and (c) network transitions.

Examples of the first transition (12 total) include a user proposing a block or a user certifying a block. There are eight transitions of the second form, such as a user delivering a next-vote for a block and advancing its period. The six network transitions include advancing the users' timers, entering and exiting a partition, and adversarial actions such as replaying a message.

To chain global states together, we define a *trace* as a non-empty sequence of global states, where each adjacent pair belongs to the global transition relation, i.e. if state g_i is the ith state in the sequence and g_{i+1} is the $(i+1)$th state, then the pair g_i and g_{i+1} belongs to the global transition relation.

By specifying path properties as propositions on traces, we are able to define generically what the property is without having to assume a fully concrete initial state. Any conditions required for the property to hold can be specified as constraints on the states of the trace being considered.

For example, the property that a block was certified in a period is captured by the following proposition:

```
Definition certified_in_period trace (r p:nat) (v:Value) :=
  ∃ (certvote_quorum : {fset UserId}),
  certvote_quorum `<=` committee r p 3 ∧
  #|` certvote_quorum | >= tau_c ∧
  ∀ (voter : UserId), voter ∈ certvote_quorum →
   certvoted_in_path trace voter r p v.
```

It states that the proposition holds for a trace if there exists a large-enough quorum of users selected for cert-voting who actually published cert-votes along that trace for the given period (the proposition `certvoted_in_path`).

4 The Safety Theorem

The safety theorem says that only one block may be certified in a round, which means the blockchain will not fork. The key assumptions of the Coq proof are some properties of cryptographic self-selection. The cryptographic arguments for the correctness of cryptographic self-selection are given by Chen et al. [2, 3], but we have not formalized cryptographic primitives and results, so our development takes these properties as unproven assumptions.

The statement of the theorem says that any two certificates from the same round must be for the same block:

```
Theorem safety : ∀ (g0:GState) (trace:seq GState) (r:nat),
  state_before_round r g0 →
  is_trace g0 trace →
  ∀ (p1:nat) (v1:Value), certified_in_period trace r p1 v1 →
  ∀ (p2:nat) (v2:Value), certified_in_period trace r p2 v2 →
  v1 = v2.
```

Note the preconditions `state_before_round r g0`, which states that that initial state `g0` is far enough back in history that no user had yet taken any actions in round `r`, and `is_trace g0 trace`, which states that the trace is a valid execution trace beginning at `g0`, with all steps taken according to the global transition relation.

It is possible to end up with certificates from multiple periods of a round. Specifically, during a network partition

which allows the adversary to delay messages, if cert-vote messages are delayed enough for some nodes to advance past the period where the first certificate was produced, but these multiple certificates will still be for the same block.

The proof of the main safety theorem stated above first considers two cases, for whether the two certificates are from the same period or different periods.

The case when the certificates are in the same period is handled in the lemma `one_certificate_per_period`. This proof uses the quorum hypotheses to conclude that there is an honest node that contributed a cert-vote to both certificates, and concludes by the lemma `no_two_certvotes_in_p` which shows that an honest node cert-votes at most once in a period (this is proved by analyzing all transition steps an honest node can take).

The case of different periods is proved using an invariant, which is established in the period that produces the first certificate and keeps holding for all later periods of the same round. The invariant property is that no step of the period produced a quorum of open next-votes, and any quorum of value next-votes must be for a given value v. This property is defined as `period_nextvoted_exclusively_for`.

Lemma `certificate_is_start_of_next_period` shows that this is true of next votes from the period that produced the first certificate, Lemma `excl_enter_excl_next` shows that if this is true for one period, it is true for the next. Lemma `prev_period_nextvotes_limits_cert_vote` shows that if the invariant holds for one period then any certificate produced in the next period can only be for the same value.

Most of the proofs only need the simpler quorum hypotheses that say that any two quorums from a single step have an honest user in common. A more complicated quorum hypothesis which relates different steps is only needed for proving the lemma `certificate_is_start_of_next_period`. The intuition is that if committee members are selected approximately randomly from the full population of users, then any property which is true for all honest members of one quorum must be true for most honest users in the overall population, so any other quorum even at different steps must have at least some honest users that also have that property. But this isn't true if the property is that of being a committee member on a particular step, and we don't know how to formalize the idea of a property being unrelated to committee membership, so we assume this hypothesis only for the specific predicate we need.

The inter-quorum hypothesis is used in between the cert-voting step and that produced the first certificate any later next-voting steps in the same period, with the property being whether a user had seen a quorum of soft-votes by the time that user finished the cert-voting step. By definition of a certificate and analysis of the transitions which send cert-votes, this property holds for every honest user which contributed a cert-vote to the certificate. Then by the inter-quorum assumptions, any quorum of next-votes must include a vote from at least one honest user which had seen a quorum of soft-votes before making their next-vote. By examining possible transitions this user couldn't make an open next-vote and could only next-vote for the value which received a certificate, so the invariant is established.

To prove the latter two lemmas `excl_enter_excl_next` and `prev_period_nextvotes_limits_cert_vote` we consider that any quorum of next-votes or cert-votes contains an honest voter, and looking at the preconditions for an honest user to make a cert-vote or next-vote. In either case making such a vote for a value besides v or making an open next-vote requires that the honest user had received a quorum of soft-votes for another value or a quorum of open next-votes from the previous period. The latter is ruled out because the invariant says no such quorum of votes was sent, and lemma `no_bottom_quorums_during_from_nextvoted_excl` contains the argument that therefore no such quorum of votes could be received. Lemma `no_bottom_quorums_during_from_nextvoted_excl` states that any quorum of soft-votes must be votes for value v if the invariant for value v held in the previous period. It is proved by analyzing the transitions which move a user to a new period to show that all honest users get v as their "starting value", and analyzing the transitions which send soft-votes to show that this starting value and the absence of quorums of open next-votes means any honest soft-vote can only be for value v.

By the overall inductive argument, we then have the safety theorem that the protocol cannot fork.

5 Conclusion

In this report, we described a complete model of the Algorand consensus protocol in Coq. We also outlined the statement and machine-checkable proof of the asynchronous safety theorem, a key property of the Algorand protocol, while also highlighting the assumptions under which the theorem holds. Our formal proof of safety (along with other lemmas leading up to it) gives additional assurance in the correctness of the protocol.

Concerning future efforts, the model we developed as part of this effort is generic in that it captures the dynamics of the Algorand consensus protocol in a way that is orthogonal to the properties that we verify about it. This means that the model can be readily used to verify other properties of the protocol beyond asynchronous safety, including most importantly *liveness* [3]. In fact, we anticipate that many of the smaller results shown about the protocol and used in the proof of safety will also constitute essential ingredients of a liveness proving effort. Nevertheless, proving liveness will likely require showing many additional results, especially related to timely message delivery and network partitioning, some of which were not needed for the safety argument. Future work also includes connecting our model to verified cryptographic primitives and results.

Acknowledgments

We thank Jing Chen, Nickolai Zeldovich, and Victor Luchangco from Algorand for their help throughout the project. This work was funded by Algorand.

References

[1] Yves Bertot and Pierre Casteran. 2004. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer Berlin Heidelberg, Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-07964-5

[2] Jing Chen, Sergey Gorbunov, Silvio Micali, and Georgios Vlachos. 2018. Algorand Agreement: Super Fast and Partition Resilient Byzantine Agreement. Cryptology ePrint Archive, Report 2018/377. https://eprint.iacr.org/2018/377.

[3] Jing Chen and Silvio Micali. 2019. Algorand: A secure and efficient distributed ledger. *Theoretical Computer Science* 777 (2019), 155–183. https://doi.org/10.1016/j.tcs.2019.02.001

[4] Cyril Cohen. 2019. A finset and finmap library. https://github.com/math-comp/finmap

[5] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (1985), 374–382. https://doi.org/10.1145/3149.214121

[6] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. 2017. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *European Conference on Computer Systems*. 328–343. https://doi.org/10.1145/3064176.3064183

[7] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. 2009. Packaging Mathematical Structures. In *Theorem Proving in Higher Order Logics*. 327–342. https://doi.org/10.1007/978-3-642-03359-9_23

[8] Herman Geuvers. 2009. Proof assistants: History, ideas and future. *Sadhana* 34, 1 (2009), 3–25. https://doi.org/10.1007/s12046-009-0001-5

[9] Georges Gonthier and Assia Mahboubi. 2010. An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning* 3, 2 (2010), 95–152. https://doi.org/10.6092/issn.1972-5787/1979

[10] John Harrison, Josef Urban, and Freek Wiedijk. 2014. History of Interactive Theorem Proving. In *Computational Logic*. Handbook of the History of Logic, Vol. 9. North-Holland, 135–214. https://doi.org/10.1016/B978-0-444-51624-4.50004-6

[11] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

[12] Mathematical Components Team. 2019. Mathematical Components Project. https://math-comp.github.io

[13] Silvio Micali. 2019. Algorand's Core Technology. https://medium.com/algorand/algorands-core-technology-in-a-nutshell-e2b824e03c77

[14] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. http://www.bitcoin.org/bitcoin.pdf

[15] George Pîrlea and Ilya Sergey. 2018. Mechanising blockchain consensus. In *Certified Programs and Proofs*. 78–90. https://doi.org/10.1145/3167086

[16] Robert Pollack. 1998. How to Believe a Machine-Checked Proof. In *Twenty Five Years of Constructive Type Theory*, G. Sambin and J. Smith (Eds.). Oxford University Press.

[17] Vincent Rahli, Ivana Vukotic, Marcus Völp, and Paulo Esteves-Verissimo. 2018. Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq. In *Programming Languages and Systems*. 619–650. https://doi.org/10.1007/978-3-319-89884-1_22

[18] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2018. Programming and Proving with Distributed Protocols. *PACMPL* 2, POPL (2018), 28:1–28:30. https://doi.org/10.1145/3158116

[19] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. 2016. Planning for Change in a Formal Verification of the Raft Consensus Protocol. In *Certified Programs and Proofs*. 154–165. https://doi.org/10.1145/2854065.2854081

[20] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Programming Language Design and Implementation*. 283–294. https://doi.org/10.1145/1993498.1993532