Formalizing Nakamoto-Style Proof of Stake

Søren Eller Thomsen and Bas Spitters

Concordium Blockchain Research Center, Aarhus University, Denmark
{sethomsen, spitters}@cs.au.dk

July 23, 2020

Abstract

Fault-tolerant distributed systems move the trust in a single party to a majority of parties participating in the protocol. This makes blockchain based crypto-currencies possible: they allow parties to agree on a total order of transactions without a trusted third party. To trust a distributed system, the security of the protocol and the correctness of the implementation must be indisputable.

We present the *first* machine checked proof that guarantees both safety and liveness for a consensus algorithm. We verify a Proof of Stake (PoS) Nakamoto-style blockchain (NSB) protocol, using the foundational proof assistant Coq. In particular, we consider a PoS NSB in a synchronous network with a static set of corrupted parties. We define execution semantics for this setting and prove chain growth, chain quality, and common prefix which together implies both safety and liveness.

1 Introduction

A Byzantine Agreement [LSP82] (BA) protocol allows a group to agree on a decision, even when some of its members behave dishonestly. Such a protocol is required to satisfy

Safety all honest parties reach the same decision;

Liveness a decision is reached eventually.

This problem naturally extends to agreeing multiple times (multi-shot-consensus or just consensus). Until 2008, the main algorithmic approach for achieving consensus was to collect a majority of votes on a decision before progressing to taking the next decision. We will refer to protocols based on this design as quorum-based protocols.

In 2008 Nakamoto's Bitcoin protocol [Nak08] revolutionized the field by introducing a fundamentally different approach for solving the problem. Instead of letting parties agree on each step of progress by multiple rounds of communication between them, Nakamoto introduced a simple protocol where parties probabilistically take turns making individual progress and disseminating this to all other parties. If parties often enough have time to see what other parties have disseminated before they make progress, such a protocol guarantees safety and liveness up to a negligible probability of failure.

The protocol works by letting all parties maintain an order-preserving data-structure over previous decisions (a block tree) and run a "lottery" to decide who is allowed to append the next block to an existing chain in the block tree. Whenever there is a winner of the lottery, they produce a block and disseminate it to all other parties. Parties receiving a block will perform a

series of checks to guarantee that the block is valid and that the party that produced the block actually won the lottery; if all the checks are correct, the parties should append the new block to their local block tree. A party will consider a slightly pruned version of their current longest valid chain to be the ordering of blocks agreed upon. We call a protocol with a similar shape, regardless of lottery mechanism, a Nakamoto-style Blockchain (NSB).

Because parties probabilistically make individual progress without waiting for a quorum, the lottery needs to be configured in such a way that the time between winners of the lottery must be long enough for blocks to propagate between parties. NSBs are therefore *only* secure in a *synchronous network* [DLS88], where an upper bound on the time it takes to deliver a message is known. Traditional quorum-based algorithms can be designed such that they are secure in either a synchronous network or a partially-synchronous network [DLS88] where there exists an unknown upper bound on message delivery time. The latter requires stronger honesty assumptions.

Nakamoto's original protocol was based on a lottery that assumes that the majority of the computing power participating in the protocol behaves honestly. The lottery functions by requiring that for a message (block) to be considered valid, the hash of the message needs to be less than a certain threshold. To participate in the lottery parties will, therefore, try to append different numbers to the messages they want to send, until they find a number which when appended to the message gives a hash which is less than the threshold. Such a lottery is called a Proof of Work lottery (PoW). Unfortunately, this design comes with a high power consumption to provide a secure protocol, as honest parties need to "mine" more valid messages than dishonest parties to ensure safety. This problem is solved by the introduction of a Proof of Stake (PoS) lottery, where parties instead can prove that they have the right to create a message for a particular round with their signature. This construction requires that the majority of stake (for some deterministic calculation of "stake") in the system behaves honestly.

Because consensus protocols are distributed, they are notoriously difficult to prove correct. In fact, some protocols were claimed to be both safe and live and passed peer review, but were later found to be either only safe or only live [AGGM+17].

Our work establishes both safety and liveness of a PoS NSB. To make our proofs indisputable we model a PoS NSB protocol similar with an abstract lottery, provide precise execution semantics for this, and reduce our proofs of safety and liveness for this protocol all the way to the axioms of mathematics using the Coq proof assistant [Tea20]. The formalization can be found at

`https://github.com/AU-COBRA/PoS-NSB`.

The formalization uses Coq 8.11.2 with mathcomp 1.11.0 [GM10], finmap 1.5.0 and coq-equations 1.2.2 [SM19]. The mathematical components (mathcomp) library has been used to formalize large parts of mathematics. It introduces a particular proof style that scales well to large developments and revolves around small-scale-reflection, which we also use for this formalization.

1.1 Contributions

We implement the behavior of honest parties participating in a PoS NSB with an abstract lottery in Coq. We use this to define the semantics of the execution of the protocol in a synchronous network accounting for the case when a static subset of parties behaves dishonestly. We prove that the protocol is *both* safe and live assuming appropriate conditions on the hash-function and the lottery, and restrictions on an adversary's capability to produce honest signatures. Safety requires at least $\frac{2}{3}$ of the stake in the protocol to be honest, whereas liveness requires only $\frac{1}{2}$ honesty.

In particular, we contribute with the following:

1. We provide the first formalization of *any* consensus algorithm that ensures both safety and liveness in a Byzantine setting. In particular, we verify that a PoS NSB protocol with an abstract lottery provides consensus. In order to do so, we provide precise semantics for executions of distributed protocols with statically corrupted parties in a *synchronous network*.

2. Beyond the mere formalization, we develop a methodology for verifying protocols by abstract functional interfaces, rather than specific non-optimized implementations. We construct a proof for a protocol that allows participants to run different concrete implementations and thus bridges the gap between our proof and a running protocol. This methodology applies both to pen-and-paper proofs as well as formalizations.

1.2 State of the Art

To provide context for this work, we give an overview of the state of the art. First, we provide an overview of analysis of NSBs and next an overview of existing mechanized proofs for consensus algorithms. In Section 6 we provide a broader comparison to other related work.

1.2.1 NSB Analysis

The first cryptographic analysis of a PoW NSB [GKL15] proved that the protocol underlying Bitcoin satisfies both safety and liveness. In order to do so they introduced the properties *chain-quality*, *chain-growth* and *common-prefix*, which together imply both safety and liveness. Their foundational analysis has been extended in several directions: The security has been analyzed in the UC-model [BMTZ17] and the analysis has been modified to cover variations of how the best chain is selected with improved properties [KMM+20]. Ren [Ren19] simplifies the original analysis.

In a PoW lottery, a winning event is tied to a specific block, which means that only the particular block that with a hash lower than the threshold will be considered valid by honest players. In PoS, however, a winning event corresponds to a party being able to sign a block that will be considered valid, which means that nothing prevents an adversary from signing multiple different blocks. Due to this attack vector a PoS protocol is inherently more difficult to analyze.

The first analysis made for a PoS NSB, was for a lottery with a unique winner in each round [KRDO17], which was followed up by an analysis of a lottery that allowed for multiple winners in each round and was generalized to a weaker network model [DGKR18]. Similar analysis have later been performed in a composable framework [BGK+18] and the bounds have been improved [BKM+20].

This work formalizes an analysis similar to previous PoW analysis, but adapts these to work for a PoS lottery. Our proof roughly follows the proofs in [KMM+20], which in order to analyze different rules for selecting the best chain rule, stated their analysis with a clear separation of necessary conditions on the lottery and combinatorial arguments. The main difference between our proofs and their proofs is in proof of common prefix property, where the difference between a PoS and a PoW protocol really becomes clear. Our proof revolves around the fact that the block corresponding to an adversarial lottery ticket can appear at most once on each chain, whereas their proof revolved around that an adversarial block can appear at most once across all chains. This implies that our proof for common prefix requires $\frac{2}{3}$ of the stake to be honest. A $\frac{1}{2}$ honesty bound can be obtained for PoS protocols [KRDO17, DGKR18, BKM+20] by more complicated proofs revolving around the notion of *characteristic strings*.

1.2.2 Formalization of Consensus Protocols

Table 1 provides an overview of selected previous formalizations of consensus algorithms in Coq.

Formalization of NSBs Toychain [PS18] was the first verification effort towards formal guarantees for any NSB (in particular a PoW NSB). They defined a relation on global states and proved basic properties about the reachable global states. In a partially synchronous network, they proved that if the system ends up in a state where no messages are waiting to be delivered, then all clients agree on the current best chain. Although that is an important property of the system it is not enough to argue about how the tree of blocks evolves when the protocol is run, as it will probably never be the case that there are no messages in transit (messages sent but not yet delivered). Toychain did not consider any Byzantine behavior and only focused on functional correctness.

Our work takes the same approach as taken in Toychain, by defining a relation on reachable global states and proving properties for these reachable states. We do, however, model a synchronous network instead of a partially synchronous one, in which stronger properties hold.

Toychain has been extracted and connected to OCaml-code [Pîr19], which provides an executable node with formal guarantees. Kaizen [KPM+19] extends the statements proven in [PS18] to apply for an actual performant implementation of a NSB through a series of refinements and transformations of the original code-base, at the cost of a slightly larger trusted computing base. This work does, however, not improve on the statements proven in [PS18].

Probchain [GS19] aims to formalize the analysis from [GKL15], but they state that their proofs are unfinished.

Formalization of quorum-based consensus Traditional (quorum-based) Byzantine fault-tolerant (BFT) consensus algorithms are also used for blockchains. Velisarios [RVVV18] is a general framework for formally proving quorum-based BFT algorithms secure in Coq. They prove a safety property of a widely used BFT algorithm, PBFT [CL99], but do not prove liveness.

A formalization of the Algorand consensus protocol [ACL+19] verifies safety of their BFT algorithm. Their proof revolves around a transition relation on global states, modelling a partially synchronous execution of the protocol.

Ethereum is planning to use a BFT algorithm as a finality layer. The Casper finality layer has been formally proven to achieve its safety property [NJH19] in the Isabelle proof assistant. In Coq, Casper has been proven to be both safe and *plausible live* [PGP+18]. Plausible live is a weaker form of liveness that ensures the protocol will never deadlock. This result was extended to also cover the revised protocol Gasper which works with a dynamic set of validators [ALP+20]. The results are proven with an abstract model of quorums on a set of messages without explicitly defining honest behavior and communication.

Formalization	**Type**	**Network**	**Safety**	**Liveness**
Toychain [PS18]	PoW NSB	Partially synchronous	(÷)	÷
Velasarios [RVVV18]	Quorum-based	Partially synchronous	✓	÷
Algorand [ACL+19]	Quorum-based	Partially synchronous	✓	÷
Gasper [ALP+20]	Quorum-based	No execution semantics	✓	(÷)
This work	PoS NSB	Synchronous	✓	✓

Table 1: Overview of previous formalizations in Coq. The formalization of Gasper does not provide execution semantics for the protocol, and so no network-model appears in their formalization.

1.3 Paper Outline

The remainder of the paper is organized as follows. Section 2 describes our notation and conventions. Section 3 describes how a PoS NSB functions. In Section 4 we will introduce the formal setting for our protocol, present the requirements for an implementation of a blocktree, define honest and adversarial behavior, and finally define *reachable* global states. Section 5 will present our general results including both the formal theorems and intuition behind the formal proofs. Section 6 will relate this work to previous work on formalizing distributed systems. Finally, Section 7 concludes.

2 Notation

The set of natural numbers is denoted $\mathbb{N} = \{0, 1, 2, \ldots\}$ and boolean values are denoted $\mathbb{B} = \{\top, \bot\}$. We adopt conventions from mathcomp and let EQTYPE be a type with decidable equality and FINTYPE be a type with a finite duplicate free enumeration.

A record type with the fields `a` and `b` of type $\mathbb{N}$ is defined by NATPAIR $:= \{\mathtt{a} : \mathbb{N}, \mathtt{b} : \mathbb{N}\}$.

SEQ T is the type of lists of type T. $[::]$ denotes the empty sequence, $[:: x]$ the list with the single element x and $+\!\!+$ the concatenation operator. We overload standard set notation for filtering and cardinality of sets to also apply to sequences. We inherit notation from mathcomp and let $=_{\mathsf{i}}$ be the property of two sequences having the same members.

We will use `teletypefont` for functions and variable names and SMALL CAPITALS for types. CamelCase (capitalized) names are used for parameters of the formalization and types whereas snake_case is used for constructs explicitly defined within the formalization.

FileName.v 》term_name are clickable links that directs to the formal definition of the described concept.

3 The Protocol

We consider a static stake PoS NSB protocol similar to the one in [DGKR18]. This section provides an informal description of the protocol, such that the description of the formal model and the exact behavior of honest parties presented in Section 4 can be guided by intuition.

We discretize time into *slots* which we assume to be totally ordered: SLOT $\triangleq \mathbb{N}$. Each party has access to a clock they can query for the current slot, a flooding network they can use to flood messages to each other, and a lottery functionality they can query to check if they are the winner of a slot. We say that a party that wins the lottery for a slot is a *baker* of this slot. Blocks contain a slot number, a hash of the predecessor, a identifier of the baker, and a signature. These are the content of messages send through the flooding network in the protocol.

Each party maintains a block tree that initially only contains a single block called the *Genesis Block*. When a block b is added to the block tree it will be added as a successor to the block in the tree with a hash that matches the predecessor of b. A path originating at the Genesis Block in a block tree is called a chain.

The protocol proceeds in slots where each party will do the following for a slot:

1. Collect all previous blocks that they have received since the last round through the flooding network and add these to their block tree if the signature is valid and the identifier of the block corresponds to a winning party for the round.

2. Evaluate the lottery to check if they are a winner of this round. If they win this slot they will:

(a) Calculate what their current longest chain is (disregarding blocks with a higher slot number than the current slot)[1]. If there are multiple longest chains of equal length they will use a tie-breaker of their choice to determine the one they consider the best[2].
(b) Create a new block that will include a hash to the head of their best chain, the current slot, their identity, and their signature.
(c) Flood this new block using the flooding network.

The protocol ensures that the participants of the protocol will agree on the current longest chain when removing a few blocks from the head of this chain. It is for the chains calculated in this way we wish to ensure both safety and liveness. Specifically, we want to ensure that the best chain of any party grows (chain growth), that honest blocks regularly are appended to this chain (chain quality), and that this chain is both consistent among parties and persistent when the protocol progresses (common prefix).

4 Formal Model

We model the protocol described in Section 3 in a synchronous network with a static but active adversary. This section describes in detail how this translates to the formal setting in which we prove our results. First, we present the basic constructs and parameters of our protocol. Next, we introduce the abstraction and specification of the block tree. Then we move on to describe our specification of the actual protocol i.e., how honest parties should behave, the global state of the entire system, and the formalization of the synchronous network. Finally, we put this together to define a relation on what state are reachable from the initial state when running the protocol with a fixed set of parties and a static but active adversary.

We will use this definition of reachable states extensively in Section 5, as we quantify all of our main statements over reachable states.

4.1 Parameters and Basic Constructs

The complete list of parameters for the formalization can be found in Figure 1.

Our model is parameterized over a type: $\textsc{Party} : \textsc{finType}$, that represents a unique identifier for a party[3] and a type $\textsc{Hash} : \textsc{eqType}$ that represents the co-domain of a hash function for blocks, `HashBlock`, which the model is parameterized over. A block, $\textsc{Block}$, is defined to be a record containing three fields

$$\textsc{Block} \coloneqq \{\texttt{pred} : \textsc{Hash},\ \texttt{slot} : \textsc{Slot}, \texttt{bid} : \textsc{Party}\}.$$

A block contains the predecessor of the block, `pred`, a baker-identifier, `bid` and a slot number in which the block was created `slot`. A *chain* is a sequence of blocks $\textsc{Chain} \triangleq \textsc{seq}\ \textsc{Block}$.

Lottery Our model is further parameterized by a predicate, $\texttt{Winner} : \textsc{Party} \to \textsc{Slot} \to \mathbb{B}$, that allows to check if a particular party has the right to create a block in a specific slot. This abstraction is intended to capture a lottery similar to the one proposed in the static-stake version of Ouroboros Praos [DGKR18], which determines the winning probability of each party based on this party's stake in the system.

We do not model signatures. In a practical PoS lottery one needs a mechanism to ensure that only the winning party can evaluate the lottery to produce a proof that this party in fact

[1] Adversarial parties might choose to evaluate the lottery ahead of time and send these to honest parties.
[2] This tie-breaker is insignificant for the security of the protocol.
[3] We make this a finite type as there as a finite supply of IP-addresses.

$$
\begin{aligned}
\textsc{Party} &: \textsc{finType} \\
\texttt{InitParties} &: \textsc{seq Party} \\
\texttt{TreeTypeMap} &: \textsc{seq Party} \to \textsc{treeType} \\
\textsc{Hash} &: \textsc{finType} \\
\textsc{Tree} &: \textsc{treeType} \\
\texttt{GenesisBlock} &: \textsc{Block} \\
\texttt{HashBlock} &: \textsc{Block} \to \textsc{Hash} \\
\texttt{Winner} &: \textsc{Party} \to \textsc{Slot} \to \mathbb{B} \\
\texttt{Honest} &: \textsc{Party} \to \mathbb{B} \\
\textsc{AdversarialState} &: \textsc{Type} \\
\texttt{AdversarialState}_0 &: \textsc{AdversarialState} \\
\texttt{AdversarialRcv}, \texttt{AdversarialBake} &: \textsc{Slot} \to \\
&\quad \textsc{Messages} \to \\
&\quad \textsc{MsgTuples} \to \\
&\quad \textsc{AdversarialState} \to \\
&\quad (\textsc{seq} \ (\textsc{Message} * \textsc{DelayMap}) * \textsc{AdversarialState})
\end{aligned}
$$

Figure 1: The complete list of parameters for the development.

is the winner of the lottery. Instead, we quantify our theorems in Section 5 by an appropriate hypothesis on the unforgeability of blocks produced by honest players (Definition 10). Neither do we model that the lottery cannot be evaluated to far into the future[4].

Valid chains Our protocol has an initial block, `GenesisBlock` : Block, that all chains should end in and which we assume to have an honest baker identifier and the slot set to 0. Using the lottery abstraction we define a valid chain.

Definition 1 (Valid chain). BlockTree.v ≫ valid_chain
We say that a chain is a valid chain if it fulfills the following three requirements

- All blocks in the chain need to be valid. A block b is valid if `Winner` (`bid` b)(`slot` b) $= \top$.
- The chain should be linked correctly: the field `pred` of a block contains a hash that is equal to that of the predecessor in the chain and the chain ends in the `GenesisBlock`.
- The projection of the fields `slot` from the chain forms a strictly decreasing sequence of slots.

We define `valid_chain` : Chain $\to \mathbb{B}$, as a computable predicate ensuring these properties are fulfilled.

4.2 BlockTree

A NSB maintains a *correct* tree of currently received blocks, from which the current *best chain* can be derived.

[4]In practice this is a desirable property, because it prevents an adversary from predicting a sequence of slots that they win.

In cryptography, it is customary to present a security analysis for specific implementations of the core data-structures used within a protocol. This is also the approach taken in previous analysis of NSB protocols [GKL15, BMTZ17, KRDO17, PS18], they provide an explicit algorithm for calculating the best chain from a set of chains and prove the security of this construction. Unfortunately, this approach creates a gap between the security analysis on an easily verifiable algorithms and the highly optimized code that is running in typical implementations of such a protocol. The main performance bottleneck of the extracted implementation of Toychain [Pîr19] is their blocktree which runs in $O(n^4)$-time. Comparable non-verified implementations run in $\sim O(n)$-time when the cost is amortized.

In this work, we take a different approach and specify the minimal requirements of a correct blocktree rather than providing an explicit construction for this data-structure.

Following the style of mathcomp we define a type, TREETYPE, that denotes a type that satisfies the requirements to achieve our security in the protocol.

Correctness conditions for a block tree For a type $\mathrm{T} : \textsc{Type}$ to be a TREETYPE, we demand that the following functions should be defined.

$$\begin{aligned}
&\texttt{tree}_0 : \mathrm{T} \\
&\texttt{extendTree} : \mathrm{T} \to \textsc{Block} \\
&\texttt{allBlocks} : \mathrm{T} \to \textsc{seq Block} \\
&\texttt{bestChain} : \textsc{Slot} \to \mathrm{T} \to \textsc{Chain}
\end{aligned}$$

The function $\texttt{tree}_0$ corresponds to the requirement that there is an initial tree that the protocol can be instantiated with, extendTree gives a way to extend any tree returning a new tree, allBlocks should give a set of blocks that the tree has been extended with and finally bestChain allows one to extract what is currently the best chain of the tree with respect to a slot.

$T : \textsc{Type}$ is a TREETYPE if is instantiated, extendable, valid, optimal and self-contained:

Definition 2 (Instantiated). BlockTree.v 〉 all_tree0

A type T is *instantiated* if no blocks are recorded in the initial structure except for GenesisBlock i.e.,

$$\texttt{allBlocks}\ \texttt{tree}_0 =_{\mathsf{i}} [:: \texttt{GenesisBlock}]\,.$$

Definition 3 (Extendable). BlockTree.v 〉 all_extend

A type T is *extendable* if extending the structure with a block is recorded properly in the set of contained blocks i.e.,

$$\forall (t : \mathrm{T})(b : \textsc{Block}), \texttt{allBlocks}(\texttt{extendTree}\ t\ b) =_{\mathsf{i}} \texttt{allBlocks}\ t \mathbin{+\!\!+} [:: b]\,.$$

Definition 4 (Valid). BlockTree.v 〉 best_chain_valid

A type T is *valid* if the best chain achieved from this structure is always a valid chain i.e.,

$$\forall (t : \mathrm{T})(sl : \textsc{Slot}), \texttt{valid_chain}\ (\texttt{bestChain}\ sl\ t).$$

Definition 5 (Optimal). BlockTree.v 〉 best_chain_best

A type T is *optimal* if the best chain less than a slot achieved from this structure is at least as good as any other chain obtained from the set of blocks recorded in the structure i.e.,

$$\begin{aligned}
&\forall (c : \textsc{Chain})(t : \mathrm{T})(sl : \textsc{Slot}), \\
&\quad \texttt{valid_chain}\ c \to c \subseteq \{b \in \texttt{allBlocks}\ t \mid \texttt{slot}\ b \le sl\} \to |c| \le |\texttt{bestChain}\ sl\ t|.
\end{aligned}$$

Definition 6 (Self-contained). BlockTree.v ⟩ best_chain_in_all

A type T is *self-contained* if the best chain less than a slot achieved from this structure is a subset of the recorded blocks in the structure i.e.,

$$\forall (t : \mathrm{T})(sl : \textsc{Slot}), \texttt{bestChain}\ sl\ t \subseteq \{b \leftarrow \texttt{allBlocks}\ t \mid \texttt{slot}\ b \leq sl\}.$$

Our development is parameterized over a specific implementation of such a type, $\textsc{Tree} : \textsc{treeType}$ that we use to build a particular tree, consisting of all blocks honest parties have received.

4.3 Parties

We represent the knowledge of a participating party as a record containing their identity, a $\textsc{treeType}$, and a blocktree of that type:

$$\textsc{LocalState} \coloneqq \{\texttt{id} : \textsc{Party},\ \texttt{tT} : \textsc{treeType},\ \texttt{tree} : \mathrm{tT}\}.$$

We further parameterize our development by a tree implementation for each party, $\texttt{TreeTypeMap} : \textsc{Party} \rightarrow \textsc{treeType}$.

This implies that our results in Section 5 are not just are quantified over all parties using a $\textsc{treeType}$. Instead, they are quantified over parties using different implementations of the same data-structure. This is a realistic scenario for a blockchain protocol, as parties might participate in the protocol with different devices and as a consequence different implementations optimized for their particular device.

Being able to make this quantification is another benefit of our abstract characterization of the core data-structure for the protocol.

Honest behavior The behavior of an honest party is defined by two stateful functions: One that defines an honest party's reaction when receiving a sequence of messages in a slot, `honest_rcv`, and one that defines what an honest party should do when baking for a slot, `honest_bake`. Both functions take an argument of type $\textsc{LocalState}$ and return an updated state together with a sequence of messages that the party wishes to flood to other parties.

$$\begin{aligned}&\texttt{honest_rcv} : \textsc{Messages} \rightarrow \textsc{Slot} \rightarrow \textsc{LocalState} \rightarrow (\textsc{Unit} * \textsc{LocalState})\\&\texttt{honest_bake} : \textsc{Slot} \rightarrow \textsc{LocalState} \rightarrow (\textsc{Messages} * \textsc{LocalState})\end{aligned}$$

The honest parties receives in a straightforward manner, as they will simply extend their blocktree with all blocks they receive, using the `extendTree`-function defined for their blocktree implementation. When an honest party is invoked to bake they will test if they are the `Winner` of the current slot. If they are the winner, they will calculate the best chain from their current block tree disregarding blocks from future slots, and create a new block with the predecessor set to the hash of the head of the best chain. Finally, they will extend their blocktree with this new block and create a message containing this block which they will flood to all other parties.

The honest behavior is computable, and to run the protocol these two functions could be extracted and connected to a network-shim[5] and a time-shim[6], similarly to what has been done for previous formalizations [KPM+19, Pîr19].

[5] Code that floods messages as well as receives messages from other parties and invoking the `honest_rcv`.
[6] Code that invokes the `honest_bake` each time a new slot start.

Adversarial parties We explicitly model an adversary within the system, by parameterizing the development by a type, $\textsc{AdversarialState}$ that the adversary can choose freely. We furthermore let the adversary choose the behavior of any corrupted party by again parameterizing our development over two functions corresponding to the adversarial behavior when receiving blocks and when baking for a slot.

$$
\begin{aligned}
\texttt{AdversarialRcv}, \texttt{AdversarialBake} : \; & \textsc{Slot} \to \\
& \textsc{Messages} \to \\
& \textsc{MsgTuples} \to \\
& \textsc{AdversarialState} \to \\
& (\textsc{seq}\ (\textsc{Message} * \textsc{DelayMap}) * \textsc{AdversarialState})
\end{aligned}
$$

This is similar to how an active adversary previously has been modeled in `Coq` [PM15, GS19]. Even though the type-signatures for `AdversarialRcv` and `AdversarialBake` are similar, we parameterize our development by two distinct functions in order to provide the adversary with as much power as possible.

The adversary's functions take more arguments than the corresponding honest ones. We model an adversary as powerful as possible and thus provides him with a complete view of the state: the entire history of messages sent in the system and those that are sent but not yet delivered. This is known as a *rushing* adversary. We also allow the adversary to supply an additional argument (of type $\textsc{DelayMap}$) to the messages he wishes to be sent, which allows him more fine-grained control of when his messages will be delivered (see Section 4.5 for details).

4.4 Global state

We define a record type $\textsc{GlobalState}$ that contains all the information for this protocol when it is executed. The $\textsc{GlobalState}$ record has the following fields.

Clock: The current slot of the system.

Message buffer: A buffer containing all messages that have been sent but not yet delivered in the system.

State map: A partial map of type $\textsc{Party} \to \textsc{option}\ \textsc{LocalState}$ that keeps track of the local state of all participating parties.

History: The history of all messages that have been sent. This is merely a book-keeping tool for describing assumptions such as the absence of hash-collisions in the state. Examples of how this is used can be found in Section 5.

Adversarial state: The adversaries state.

Execution order: The order in which the system should activate its parties. This is merely a bookkeeping allowing the environment to decide the order of activations (see Section 4.6).

Progress: The progress that the system has made within a single slot

$$\textsc{Progress} \triangleq \{\texttt{Ready}, \texttt{Delivered}, \texttt{Baked}\}.$$

How a global state can change its progress is defined in Section 4.6.

4.5 Network

We assume a lock-step-synchronous network with a known upper bound on the delivery time. This is similar to what the first analysis of both PoW [GKL15] and PoS [KRDO17] assumes. This can be extended to a semi-bounded delay network (with a known upper bound) in the same way as [GKL15, DGKR18]. This network model is different from the analysis in [PS18], which assumed only a partially synchronous network[7]. However, NSBs are not secure in that model.

More precisely, we assume that time is discretized into slots which are coarse enough for honest parties to have enough time to first execute their computations for a slot and then send out messages. At this time there should be enough time left in the round such that any message sent out at this point is ready for the delivery phase of the next round. This assumption enables the possibility of creating a flooding network with the property that if a message is sent by an honest party in slot sl then it will be delivered to any other party at time $sl + 1$.

Adversarial parties sending messages in slot sl does, however, have the possibility of postponing sending their messages until the very end of the round in which case they can choose to let some honest parties receive their message in slot $sl + 1$ and others in slot $sl + 2$.

Even though the underlying assumption is similar to previous work [GKL15, KRDO17, DGKR18], this might at first seem more complicated. The reason is that in their work honest parties diffuse entire chains whenever they are mined. This allows the adversary to send different messages to each party, as if an honest party extends such a chain, the entire chain will get propagated to the other honest parties anyway.

Instead, we choose to send blocks through a flooding network, and assume that this underlying functionality relays messages when they are received. This achieves that what one honest party has received in one slot will also have been received by any other party in the next slot. This is closer to NSBs running in practice and more realistic than what previous pen-and-paper analysis has modeled.

To capture this network in our formalization, we introduce the type MESSAGE as an inductive type with only a single constructor namely `BlockMsg` : BLOCK $\rightarrow$ MESSAGE, and the record MSGTUPLE defined by

$$\text{MsgTuple} := \{\texttt{msg} : \text{Message},\ \texttt{rcv} : \text{Party},\ \texttt{cd} : \text{Delay}\},$$

where DELAY $\triangleq \{1, 2\}$. The field `msg` contains the actual message that is to be delivered at the receiving party contained in the field `rcv`. `cd` is the current delay of the message which. This field will be decremented for all messages as time progresses in the model.

The flooding network available to the parties is formalized as a set of functions that operate on a global state. The functionalities `flood_msgs` and `flood_msgs_adv` enable the honest parties, the adversary, respectively to send messages.

$$\texttt{flood_msgs} : \text{Messages} \rightarrow \text{GlobalState} \rightarrow \text{GlobalState}$$
$$\texttt{flood_msgs_adv} : \text{seq}\ (\text{Messages} * \text{DelayMap}) \rightarrow \text{GlobalState} \rightarrow \text{GlobalState}$$

Both functions will create a new message-tuple with the message for each party in the execution order of the global state. `flood_msgs` will set the delay of the messages that are being sent to 1, whereas the `flood_msgs_adv` takes an extra parameter for each message namely a DELAYMAP $\triangleq$ PARTY $\rightarrow \{1, 2\}$, such that the adversary for each message explicitly can choose what parties should have it delivered in the next round and what parties should have it delivered in two rounds.

[7] A network that only guarantees that messages eventually will be delivered.

4.6 Reachable Worlds

To be able to reason about the reachable states of the protocol, we first define an initial global state, $\texttt{N}_0$: GLOBALSTATE. To this end we parameterize our development over a sequence of parties participating in the protocol, `InitParties` : SEQ PARTY, and create an initial state for all these parties with their tree set to $\texttt{tree}_0$. The development is also parameterized over any initial state that an adversary wants to choose, $\texttt{AdversarialState}_0$: ADVERSARIALSTATE.

$\texttt{N}_0$ is now defined in a straightforward manner with no messages in the message-buffer, nothing in the history, $\texttt{AdversarialState}_0$, and the parties' respective initial states.

We also parameterize our protocol over a total map `Honest` : PARTY $\rightarrow \mathbb{B}$, which decides what function should be invoked for each respective party. This corresponds to the adversary being able to statically decide who should be corrupted.

To capture how the protocol progresses we define a relation over atomic steps of a global state that enforces a state-transition system. A depiction of the transition system can be found in Figure 2. In the definition below *progress* refers to the progress stored in a global state.

Definition 7 (Atomic step reachable). Schedule.v 〉SingleStep

For any two states N_1, N_2 : GLOBALSTATE, we say that N_2 is reachable in an atomic step from N_1 if one of the following steps are taken.

Receive: If the progress of N_1 is `Ready`, then N_1 can step to the state obtained by invoking each respective parties delivery-function, update the state of the state according to the outcome of this, and set the progress to `Delivered`.

Bake: If the progress of N_1 is `Delivered`, then N_1 can step to the state obtained by invoking each respective parties bake-function, update the state of the state according to the outcome of this, and set the progress to `Baked`.

Increment: If the progress of N_1 is `Baked`, then N_1 can step to the state obtained by incrementing the slot number and updating the progress to `Ready`.

Permute execution order: Any N_1 can step to the state obtained by permuting the execution order of N_1.

Permute message buffer: Any N_1 can step to the state obtained by permuting the message buffer of N_1.

When N_1 can step to N_2 in one atomic step, we write $N_1 \rightsquigarrow N_2$.

This transition relation can be seen as an environment activating the parties in a restricted order. We model a adversarial environment by allowing permutations of the message buffer and the execution order. This models a very powerful adversary who gets to choose the exact message order for all messages sent, and decides the execution order for each step[8]. Definition 7 is formalized as an inductive relation over global states in Coq.

We extend this definition to cover multiple steps as the reflexive transitive closure of atomic steps.

Definition 8 (Reachable). Schedule.v 〉BigStep

For any two states N_1, N_2 : GLOBALSTATE we say that N_2 is *reachable* from N_1 if N_2 is reachable in zero or more atomic step from N_1. We write $N_1 \Downarrow N_2$.

Our definition of reachable enforces that the set of parties participating in the protocol remains static through the execution of the protocol.

[8]This is also our reason for representing the execution order and message buffer as lists rather than multisets as we wish to give the adversary as much power as possible, by letting him determine the exact order.

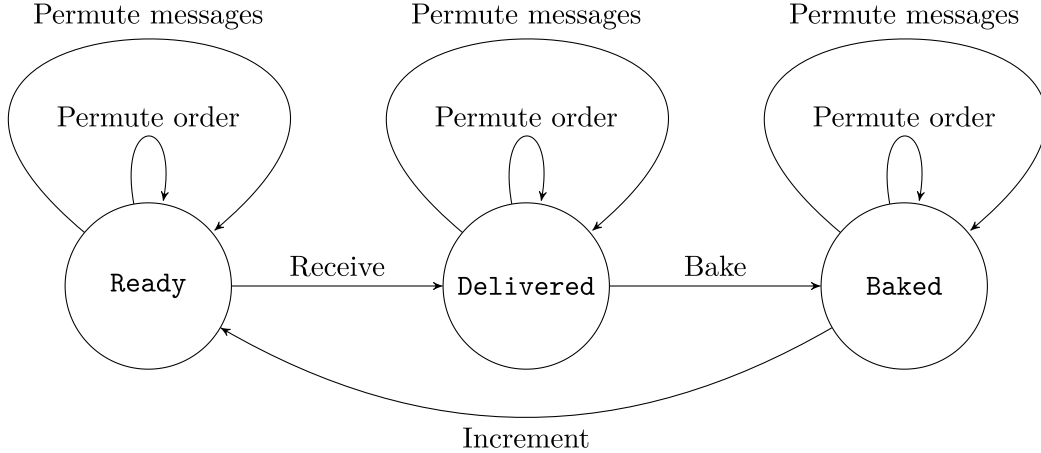

Figure 2: A depiction of the transition system that defines reachable states.

5 Safety and Liveness

The properties *chain growth*, *chain quality* and *common prefix* ensure both liveness and safety of a NSB [GKL15]. Chain growth says that the length of the best chain an honest party increases over time. Chain quality says that within a sufficiently large consecutive chunk of blocks of a best chain some of them must be honest. Common prefix says that the chains of honest parties will be a prefix of each other if we remove some blocks from the chain.

This section will describe and state these main theorems for this formalization and give an outline of the structure of each of the proofs. The entire proof amounts to roughly 6k lines of code using mathcomp's compact proof language.

Throughout the section we make two standard assumptions about the transition system. We assume that the list of parties participating (`InitParties`) in the protocol is unique, i.e., that no party will be activated twice during the same atomic step, and that there is at least one honest party among the participants[9].

Phrasing of theorems Our chain growth, chain quality and common prefix are stated as implications rather than the absolute probabilistic statements given in previous analysis. Chain growth relies only on a certain amount of lucky slots within the time-span of states, whereas chain quality relies on a collision-free state, a forging-free state and certain condition on the winning events in a time-span. Common prefix relies both on a collision-free and a forging-free state, but explicitly states the property holds or a bad event happens — namely that the adversary has gotten an advantage that he statistically cannot achieve for a large k.

A probabilistic statement can be obtained by bounding the probabilities of the desired hypotheses (or conclusion). Formalizing this dependents on the specific lottery functionality, the hash function, and the signature-scheme.

5.1 Defining Preconditions

We start by defining some basic concepts. First, we specialize hash-collisions to our setting. Next, we state an assumption on the adversary's capability to publish blocks with honest identifiers,

[9]This is not a requirement on the stake of the honest parties, but simply a requirement that at least one of the actual parties in the protocol behaves honestly. The requirements on the lottery and thus on the stake will appear as preconditions for the individual statements.

before we move on to define certain good and bad events with respect to the lottery.

Any NSB protocol only provides its guarantees under the assumption that there are no hash-collisions throughout the execution. We define this as a *collision-free* state.

Definition 9 (Collision-free). CQ.v collision_free
A global state N : GLOBALSTATE with block history bh : SEQ BLOCK is *collision-free* if

$$\forall b, b' : \text{BLOCK}, b, b' \in bh \to \texttt{HashBlock}\ b = \texttt{HashBlock}\ b' \to b = b'.$$

For any two global states N_1, N_2 : GLOBALSTATE, if $N_1 \Downarrow N_2$ and N_2 is collision-free, then N_1 is also collision-free, as block histories are monotonously growing over reachable states. We have taken care to phrase each of our main theorems using this definition, instead of assuming a global axiom on the injectivity of the hash-function or that any reachable state is collision-free. The introduction of such global axioms could, in the worst case, lead to a inconsistent logic, but it would also not be able to bound the probability that such an axiom is satisfied using a collision-resistant hash-function.

Another assumption that is needed in order to be able to state our main theorems is that the adversary cannot forge any honest blocks through the execution that led to a global state. We do not model signatures explicitly, so instead, we assume that the adversary cannot send out any block with the `bid`-field set to the identifier of an honest party that is not already a part of the block history.

Definition 10 (Forging-free). CQ.v forging_free
We say that a global state N : GLOBALSTATE is *forging-free* if for any activation of the adversarial functions, `AdversarialBake`, `AdversarialRcv` with parameters from a global state N' : GLOBALSTATE where $N' \Downarrow N$ implies that there are no honest blocks in what the adversary sends that is not already in the block history of N'.

In order to state this in the formalization, we introduce a more fine-grained refinement of the reachable transition-relation. We need this to be able to precisely state that the assumption holds in between each individual party-activation and not only in the synchronous steps.
The definition of forging-free closely corresponds to the property one could achieve by using an EUF-CMA (existential unforgeability under chosen message attack) secure signature scheme to sign blocks. If a particular global state, N : GLOBALSTATE, is not forging-free, the adversary has been able to forge a message between the initial state $\texttt{N}_0$ and N, and thus succeeded in breaking the signature scheme.

We define a lucky slot to be any slot where an honest party wins the lottery and an adversarial slot to be the corresponding concept for adversarial parties. Finally, we define honest advantage to be the difference between these two amounts over a sequence of slots.

Definition 11 (Lucky slot). CG.v lucky_slot
A slot sl is a *lucky slot* if there is a party $p \in$ `InitParties` s.t. `Winner` $p\ sl \wedge$ `Honest` p.

Definition 12 (Super slot). CP.v super_slot
A slot sl is a *super slot* if there is a exactly one party $p \in$ `InitParties` s.t. `Winner` $p\ sl \wedge$ `Honest` p.

Definition 13 (Adversarial slot). CQ.v adv_slot
A slot sl is an *adversarial slot* if there is a party $p \in$ `InitParties` s.t. `Winner` $p\ sl \wedge \neg$`Honest` p.

There is a close connection between a lucky slot and the creation of a *left-isolated block* in the analysis of PoW [KMM+20], as we have scaled our slots such that all honest blocks have time to propagate before the round begins. Similarly, super slots corresponds to *isolated* blocks. We call the block won by an honest player in a super slot a *super block*

Definition 14 (Honest advantage). CQ.v 〉 honest_advantage_range
We define the *honest advantage* for an interval of slots to be the difference between the number of lucky slots and the number of adversarial slots in this period.

5.2 Preliminary Lemmas

We now state some selected definitions and lemmas that are used to prove our main theorems. The first lemma we introduce describes how knowledge propagates between honest parties.

Lemma 1 (Honest party knowledge propagation). CG.v 〉 honest_tree_subset
Let N_1, N_2 : GLOBALSTATE and p_1, p_2 : PARTY. If $\mathtt{N}_0 \Downarrow N_1$, $N_1 \Downarrow N_2$, p_1 is a party in N_1 with tree t_1, p_2 is a party in N_2 with tree t_2, N_1 is at `Ready`, N_2 is at `Delivered`, and N_1 and N_2 are in the same slot then

$$\texttt{allBlocks}\ t_1 \subseteq \texttt{allBlocks}\ t_2.$$

Proof sketch. Our main observation is that at any point in time a block is in the tree of p_1, it is either also already in p_2's tree or to be delivered at the next delivery transition. Blocks can be added when an honest party wins the right to bake a block, in which case they will immediately send the block to all other parties and thus fulfill the invariant, or they can be added by an adversary and thereby delivered to an honest party by a delivery event, in which case it will be delivered to all other honest parties in the following delivery slot (by our network assumption).

This is in particular true when p_1 and p_2 is at `Ready`, which means that after the delivery transition p_2 will know all the blocks that p_1 knew before. □

Since honest parties extend their trees monotonously this subset-relation will also extend to any state that leads to N_1 and any state that is reachable from N_2.

The core insight of the common-prefix proofs is that each time a super-slot is won the block produced in this slot will not have the same depth in a chain as any other honest block. In order to define this precisely, we define how to calculate a chain from a block[10].

Definition 15 (Chain from a block). CP.v 〉 cfb
We define the chain from a block b : BLOCK with respect to a sequence of blocks bp : SEQ BLOCK to be the chain obtained by following the pointers to from b through bp ending in `GenesisBlock`. We write `cfb` b bp to denote this chain. If no such chain can be obtained by following pointers in bp we say that `cfb` b $bp = [::]$.

Definition 16 (Position of a block). CP.v 〉 pos
We furthermore define the *position* of a block, written `pos`, to be the length of the chain obtained by following the pointers from the block,

$$\texttt{pos}\ b\ bp := |\texttt{cfb}\ b\ bp|.$$

As this is not a structurally recursive function we use the `coq-equations` plugin [SM19] in order to automatically get a strong induction principle. This allows us to prove the following lemma that is a central step towards proving the common prefix property.

[10] This definition does not appear in previous pen and paper proofs, which only talks about positions of blocks without defining with respect to what set of blocks.

Lemma 2 (Super block positions). CP.v » no_honest_pos_share_sb
Let N : GlobalState, sb, b : Block and let bh : seq Block be the history of blocks in N. Suppose $\mathtt{N}_0 \Downarrow N$, N is forging-free and collision-free, $b, sb \in bh$, b is honest and sb is a super block then

$$\texttt{pos}\ sb\ bp \neq \texttt{pos}\ b\ bp.$$

Proof sketch. The proof proceeds by induction on the transition relation $\mathtt{N}_0 \Downarrow N$. The base case is trivial as there are no blocks in the block history of $\mathtt{N}_0$. In the induction case we distinguish between which transition was taken last.

Receive: Receiving messages does not change the subset of the block history that is honest. Moreover, a collision-free state guarantees that the positions of the honest blocks that are already in the block history do not change.

Bake: Let sl be the slot of N. We note that any honest block $b' \in bh$ must have a slot number that is less than or equal to that of the current state, and distinguish between these two cases.

$\texttt{slot}\ sb < sl$: Any honest party that bakes a new block in this step must have known about sb (by Lemma 1) and are aware of a valid chain that is at least as long as the position of sb. We will therefore have for any new block b that is baked in such a way that $\texttt{pos}\ sb\ bh < \texttt{pos}\ b\ bh$.

$\texttt{slot}\ sb = sl$: There is exactly one honest party that bakes a block in this step. By Lemma 1 this party must know about all other honest blocks baked in previous rounds. We will therefore have that for any old honest block b that $\texttt{pos}\ b\ bh < \texttt{pos}\ sb\ bh$.

Increment/Permute execution order/message buffer: These transitions do not change the block history.

□

At last we define pruning and a prefix, as well as a minor lemma relating the notions in order to phrase and prove our common prefix theorem.

Definition 17 (Pruning). CP.v » prune_time
Let c : Chain be a chain and let sl : Slot be a slot. We prune c by sl by removing all blocks that has a slot higher than sl,

$$\texttt{prune}\ sl\ c \triangleq \{b \leftarrow c \mid \texttt{slot}\ b \leq sl\}.$$

For a valid chain, pruning corresponds to simply removing blocks until the head of the chain is below a or equal to a certain slot. We finally define prefix[11].

Definition 18 (Chain prefix). SsrFacts.v » suffix
Let c_1, c_2 : Chain. We say that c_1 is a *prefix* of c_2 if there exists a c_3 : Chain such that $c_3 \mathbin{+\!\!+} c_1 = c_2$. We write $c_1 \preceq c_2$.

Lemma 3 (Prune prefix transitivity). CP.v » prune_suffix_trans
For any sl : Slot and c_1, c_2, c_3 : Chain such that $\texttt{prune}\ sl\ c_1 \preceq c_2$ and $\texttt{prune}\ sl\ c_2 \preceq c_3$, we have $\texttt{prune}\ sl\ c_1 \preceq c_3$.

[11] Technically this is a *suffix* due to the orientation of our list structure, but to avoid confusion we use the word prefix to align with previous results.

5.3 Main Theorems

We are now ready to state our three main theorems. For clarity we ignore the constants -1 and 1 when counting the number of lucky/adversarial/super slots. These constants are used to account for adversary's ability to wait one more round to bake than the honest parties, because he immediately knows of all previously baked blocks.

At a slot sl any party with a tree t will consider their best chain to be the chain calculated from the tree by disregarding all blocks from this slot and the future, `bestChain` $(sl-1)\ t$. We will show the three key properties for such chains.

The chain growth property intuitively says that in each period, the best chain of any honest party will increase at least by an amount that is proportional to the number of lucky slots in that period.

Theorem 1 (Chain Growth). CG.v ≫ chain_growth_parties
Let N_1, N_2 : GlobalState, p_1, p_2 : Party, sl_1, sl_2 : Slot and $w : \mathbb{N}$. If $\mathtt{N_0} \Downarrow N_1$, $N_1 \Downarrow N_2$, p_1 is a party in N_1 with tree t_1, p_2 is a party in N_2 with tree t_2, the round of N_1 is sl_1, the round of N_2 is sl_2 and there are at least w lucky slots between N_1 and N_2 then

$$|\texttt{bestChain}\ (sl_1 - 1)\ t_1| + w \leq |\texttt{bestChain}\ (sl_2 - 1)\ t_2|.$$

Proof sketch. We proceed by induction on the number of lucky slots, w.

The base case follows by monotone growth of honest chains over time[12]. In the induction case we identify the global state N with the lowest slot number sl s.t., $N_1 \Downarrow N$, $N \Downarrow N_2$, and `lucky_slot` sl. In the global state N, we establish that the honest party who wins the slot creates a new chain that is strictly longer than any chain of an honest party in N_1, as they knew what was there before by Lemma 1. We complete the proof by applying the induction hypothesis to N. □

We now present the chain quality property. The chain quality property intuitively says that within any consecutive chunk of blocks of an honest party's chain, a ratio proportional to the difference between the number of honest and adversarial slots will be honest in this consecutive chunk.

Theorem 2 (Chain Quality). CQ.v ≫ chain_quality
Let N : GlobalState, p : Party and $w : \mathbb{N}$. Suppose $\mathtt{N_0} \Downarrow N$, N is forging-free and collision-free, p is a party in N with tree t, the round of N is sl, and let $B_i \ldots B_j$ be a consecutive interval of blocks of `bestChain` $(sl-1)\ t$. If there is an *honest advantage* of at least w for time periods longer than `slot` $B_i -$ `slot` B_j then the amount of honest blocks in $B_i \ldots B_j$ will be at least w.

Proof sketch. We define $B_{\hat{i}}$ and $B_{\hat{j}}$ s.t. $B_{\hat{i}} \ldots B_{\hat{j}}$ is the smallest interval of `bestChain` $(sl-1)\ t$ such that $B_i \ldots B_j \subseteq B_{\hat{i}} \ldots B_{\hat{j}}$, $B_{\hat{i}}$ is honest and $B_{\hat{j}}$ is either honest or the head of `bestChain` $(sl-1)\ t$ [13]. As $B_{\hat{i}}$ is honest, we can apply Theorem 1 to establish that $|B_{\hat{i}} \ldots B_{\hat{j}}|$ is at least the amount of adversarial slots in the time span between the creation of $B_{\hat{i}}$ and $B_{\hat{j}}$ plus the honest advantage in this time span. As all blocks in a valid chain (and as `bestChain` $(sl-1)\ t$ is valid) have unique slot numbers this implies that the there must be at least w honest blocks in between $B_{\hat{i}}$ and $B_{\hat{j}}$ and therefore also w honest blocks in $B_i \ldots B_j$. □

[12] Technically, the slot number of N_1 needs to be strictly smaller than that of N_2, as the knowledge of p_1 needs to have time to propagate to p_2 by Lemma 1.

[13] $B_{\hat{i}}$ is well defined as we consider the genesis block to be honest.

We achieve the full chain-quality property that is defined for any fragment of any honest party's best chain rather than the somewhat weaker property considered in [Ren19].

Together chain growth and chain quality prove liveness, as chain growth ensures that more blocks will be appended to any honest party's log and chain quality ensures that there will be some honest input to this log. The preconditions for Theorems 1 and 2 can be shown to hold with a probabilistic bound assuming only honest majority. This is as good as the previous analysis done with pen and paper.

The common prefix property informally says that during the execution of the protocol the chains of honest parties will always be a common prefix of each other (after removing some blocks on the chain). We follow [KMM+20, GKL15] and define two variants of the common prefix property. The first variant ensures that any two best chains of honest parties are consistent within a single round, and the second variant ensures that the best chain of an honest party is consistent with earlier best chains of any honest party. The latter variant constitutes safety for blockchain consensus protocols.

Lemma 4 (Common prefix-lemma). CP.v 〉 cp_lem_gen_inc
Let N : GlobalState, p : Party, c : Chain, k : Slot and bh : seq Block. Suppose $\mathtt{N}_0 \Downarrow N$, N is forging and collision-free, p is a honest party in N with tree t, the round of N is sl, the block history of N is bh, $c \subseteq bh$, that c is a valid chain, all blocks in c have a slot number less than sl and that $|\mathtt{bestChain}\ (sl-1)\ t| \leq |c|$. Then one of the following holds:

1. $\mathtt{prune}\ k\ (\mathtt{bestChain}\ (sl-1)\ t) \preceq c$
2. There exists sl' : Slot, s.t. $sl' \leq k$ and the amount of super slots in the slot range from sl' to sl is less than two times the amount of adversarial slots in the same period of time.

Proof sketch. We define b' to be first honest block in the common stem of c and $\mathtt{bestChain}\ (sl-1)\ t$. If $k < \mathtt{slot}\ b'$ we can conclude $\mathtt{prune}\ k\ (\mathtt{bestChain}\ (sl-1)\ t) \preceq c$. Otherwise we show Item 2.

We define sl' as $\mathtt{slot}\ b'$. Let bh be the block history of N. For any honest block b that is produced between $\mathtt{slot}\ b'$ and sl, we have

$$\mathtt{pos}\ b'\ bp < \mathtt{pos}\ b\ bp \leq |\mathtt{bestChain}\ (sl-1)\ t| \leq |c|.$$

$\mathtt{pos}\ b'\ bp < \mathtt{pos}\ b\ bp$ because at the time b was created the honest party that created it knew about a chain of length $\mathtt{pos}\ b'\ bp$, and $\mathtt{pos}\ b\ bp \leq |\mathtt{bestChain}\ (sl-1)\ t|$ as otherwise there would be a longer chain available to p at time sl. Any adversarial slot can appear at most once on each chain. So, by Lemma 2 there must be an adversarial slot for every two super blocks in this period. □

Remark 1. For any reachable N global state with two honest parties, Lemma 4 can be instantiated with c being the longer of the best chains for these parties. This will thus give us that the best chain of any honest party will be a prefix of any other honest party's best chain.

Theorem 3 (Timed Common prefix). CP.v 〉 timed_common_prefix
Let N_1, N_2 : GlobalState, p_1, p_2 : Party, sl_1, sl_2 : Slot and k : Slot. If $\mathtt{N}_0 \Downarrow N_1$, $N_1 \Downarrow N_2$, N_2 is forging-free and collision-free, p_1 is a party in N_1 with tree t_1, p_2 is a party in N_2 with tree t_2, the round of N_1 is sl_1 and the round of N_2 is sl_2. Then one of the following holds:

1. $\mathtt{prune}\ k\ (\mathtt{bestChain}\ (sl_1-1)\ t_1) \preceq (\mathtt{bestChain}\ (sl_2-1)\ t_2)$

2. There exists sl', sl'' : SLOT, s.t. $sl' \leq k$, $sl_1 \leq sl'' \leq sl_2$ and that the amount of super slots in the slot range from sl' to sl'' is less than two times the amount of adversarial slots in the same period of time.

Proof sketch. The proof proceeds by induction on the transition relation $N_1 \Downarrow N_2$. The base case where $N_1 = N_2$ is solved by applying Lemma 4 (in particular Remark 1). In the induction case we distinguish between which transition was taken last.

Receive: The induction hypothesis gives us that the statement is true for the tree t_2' which p_2 has just before he receives the messages in this round. The messages that p_2 receives in this round must however already be in the block history and therefore Lemma 4 can be applied. This either results in Item 2 or we can apply Lemma 3 to achieve that $\texttt{prune}\ k\ (\texttt{bestChain}\ (sl_1 - 1)\ t_1) \preceq (\texttt{bestChain}\ (sl_2 - 1)\ t_2)$.

Bake: The induction hypothesis gives us that the statement is true for the tree t_2' which p_2 has just before he tries to bake for this slot. If p_2 bakes a block for the slot sl, the new block that is baked cannot itself be a part of the $\texttt{bestChain}\ (sl_2 - 1)\ t_2$ but it might however still change the internal structure of the t_2 such that $\texttt{bestChain}\ (sl_2 - 1)\ t_2 \neq \texttt{bestChain}\ (sl_2 - 1)\ t_2'$. This new chain must, however, already be a part of the block history, and therefore Lemma 4 can be applied. This either results in Item 2 or we can apply Lemma 3 to achieve that $\texttt{prune}\ k\ (\texttt{bestChain}\ (sl_1 - 1)\ t_1) \preceq (\texttt{bestChain}\ (sl_2 - 1)\ t_2)$.

Increment: Incrementing the time allows for a slightly longer best chain than just before time was incremented. We apply the induction hypothesis to establish the relationship between the old best chain of p_2 and the best chain of p_1. Now we again apply Lemma 4 and Lemma 3.

Permute execution order/message buffer: These transitions do not change the best chains of any honest parties and the induction hypothesis can be applied.

□

To be able to exclude that Item 2 happens by a probabilistic bound for large k one needs to assume at least $\frac{2}{3}$ of the stake is honest.

If we considered a *covert adversary*, i.e., an adversary that does not want to leave a trace that they did not follow the protocol, the adversary would only be able to place each block on one chain. This would immediately give us a tighter bound.

6 Related Work

Verified distributed systems A series of works have focused on formally verifying distributed systems in a non-Byzantine setting. Raft [OO14] is a consensus algorithm that withstands benign failures and is simpler than similar algorithms, such as Paxos. The safety property of Raft was formalized using the Verdi framework [WWP+15, WWA+16]. Verdi relies on a shallow-embedding of protocols into Coq and provides the verified-system-transformers which facilitate composable verification. Applying Coq's extraction to the Raft consensus protocol one obtains an implementation when connected to a network-shim. Their extracted code is as efficient as non-verified implementations.

Disel [SWT18] is a framework for verifying distributed systems. It is built on a foundation of separation logic embedded in Coq and allows verifying OCaml like programs using a Hoare style reasoning. One can use the partial correctness of their Hoare style specifications to reason about safety. Aneris [KJTO+20] is another framework embedded in Coq for verifying distributed

systems. It is built upon the Iris separation logic [JSS+15], which allows reasoning about multi-threaded computations for local nodes while being able to combine the statements about local nodes to safety statements for the entire system. Neither Disel nor Aneris has been used to reason about Byzantine behavior.

Lamport designed TLA+ [Lam92] with the specific purpose of formally specifying and checking distributed protocols. This was used together with the TLAPS model-checker to check the safety (but not liveness) of PBFT [Lam11].

IronFleet [HHK+15] is a framework for combining both TLA-style specifications that are machine-checkable and Hoare style specifications in Dafny [Lei10]. They prove a performant implementation of Paxos (a consensus algorithm withstanding benign failures) to be both safe and live.

Verified cryptographic protocols There are a few works that verify cryptographic protocols that are designed to be robust in an adversarial setting. We mention the formalizations of multiparty computation [HKO+18] and the AWS key-server [ABB+19] in Easycrypt. These works benefits from Easycrypt's logic that allows to reason about game-hops easily but also show limitations of Easycrypt's build-in programming language `pwhile` that lacks primitives for communication. The latter increases the complexity of the formalizations.

Modern cryptographic security proofs of consensus, e.g. [BMTZ17, BGK+18], emphasize the use of an informal composible framework. This will also be important for us when we want to prove that the system remains secure when we instantiate our lottery functionality with an implementation that has been proved to be secure in isolation. Fortunately, such modular/composible frameworks are being developed more formally [CSV19, LSBM19]. However, only very simple protocols have been proven secure using these, due to the complexity of the frameworks themselves.

7 Conclusion

We have given a formalized proof that a PoS NSB protocol with a static set of corrupted parties in a synchronous network has chain growth, chain quality, and common prefix. This has required us to define precise semantics for the execution of the protocol. We have defined honest behavior by computable functions and used this to define a relation on reachable global states. We have also developed a new methodology for specifying core data-structures by their functional behavior rather than concrete implementation. This gives a proof that can provide guarantees for actual optimized running implementations, if these simply live up to the specification. The methodology further has the consequence that we are able to prove security for parties running different implementations of the same protocol.

Acknowledgments We thank Ilya Sergey for helpful discussions on Toychain and its variants, Daniel Tschudi for discussions in the beginning of the project, Thomas Dinsdale-Young for providing helpful insights into the Concordium Blockchain implementation, Jesper Buus Nielsen for helping to clarify the cryptographic models and ideas to the simple proof of common prefix, and Sabine Oechsner for valuable discussions and feedback.

References

[ABB+19] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Matthew Campagna, Ernie Cohen, Benjamin Gregoire, Vitor Pereira, Bernardo Portela, Pierre-Yves Strub,

and Serdar Tasiran. A machine-checked proof of security for AWS key management service. In *CCS*, 2019.

[ACL+19] Musab A. Alturki, Jing Chen, Victor Luchangco, Brandon M. Moore, Karl Palmskog, Lucas Peña, and Grigore Rosu. Towards a verified model of the Algorand consensus protocol in Coq, 2019.

[AGGM+17] Ittai Abraham, Guy Golan-Gueta, Dahlia Malkhi, Lorenzo Alvisi, Ramakrishna Kotla, and Jean-Philippe Martin. Revisiting fast practical byzantine fault tolerance. ArXiv, 2017.

[ALP+20] Musab A. Alturki, Elaine Li, Daejun Park, Brandon Moore, Karl Palmskog, Lucas Pena, and Grigore Roşu. Verifying Gasper with dynamic validator sets in Coq. Technical report, 2020.

[BGK+18] Christian Badertscher, Peter Gaži, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros Genesis: Composable proof-of-stake blockchains with dynamic availability. In *CCS*, 2018.

[BKM+20] Erica Blum, Aggelos Kiayias, Cristopher Moore, Saad Quader, and Alexander Russell. The combinatorics of the longest-chain rule: Linear consistency for proof-of-stake blockchains. In *SODA*, 2020.

[BMTZ17] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In *CRYPTO*, 2017.

[CL99] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *OSDI*, 1999.

[CSV19] Ran Canetti, Alley Stoughton, and Mayank Varia. EasyUC: Using EasyCrypt to mechanize proofs of universally composable security. In *CSF*, 2019.

[DGKR18] Bernardo David, Peter Gaži, Aggelos Kiayias, and Alexander Russell. Ouroboros Praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *EUROCRYPT*, 2018.

[DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 1988.

[GKL15] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *EUROCRYPT*, 2015.

[GM10] Georges Gonthier and Assia Mahboubi. An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning*, 2010.

[GS19] Kiran Gopinathan and Ilya Sergey. Towards mechanising probabilistic properties of a blockchain. In *CoqPL*, 2019.

[HHK+15] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: Proving practical distributed systems correct. In *SOSP*, 2015.

[HKO+18] Helene Haagh, Aleksandr Karbyshev, Sabine Oechsner, Bas Spitters, and Pierre-Yves Strub. Computer-aided proofs for multiparty computation with active security. In *CSF*, 2018.

[JSS⁺15] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*, 2015.

[KJTO⁺20] Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. Aneris: A mechanised logic for modular reasoning about distributed systems. In *ESOP*, 2020.

[KMM⁺20] Simon Holmgaard Kamp, Bernardo Magri, Christian Matt, Jesper Buus Nielsen, Søren Eller Thomsen, and Daniel Tschudi. Leveraging weight functions for optimistic responsiveness in blockchains. ePrint, 2020.

[KPM⁺19] Faria Kalim, Karl Palmskog, Jayasi Mehar, Adithya Murali, Indranil Gupta, and Phalgun Madhusudan. Kaizen: Building a performant blockchain system verified for consensus and integrity. In *FMCAD*, 2019.

[KRDO17] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *CRYPTO*, 2017.

[Lam92] Leslie Lamport. Hybrid systems in TLA+. In *Hybrid Systems*, 1992.

[Lam11] Leslie Lamport. Byzantizing Paxos by refinement. In *DISC*, 2011.

[Lei10] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, 2010.

[LSBM19] Andreas Lochbihler, S. Reza Sefidgar, David A. Basin, and Ueli Maurer. Formalizing constructive cryptography using CryptHOL. In *CSF*, 2019.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 1982.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[NJH19] Ryuya Nakamura, Takayuki Jimba, and Dominik Harz. Refinement and verification of CBC Casper. In *CVCBT*, 2019.

[OO14] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Annual Technical Conference*, 2014.

[PGP⁺18] Karl Palmskog, Milos Gligoric, Lucas Pena, Brandon Moore, and Grigore Roşu. Verification of Casper in the Coq proof assistant. Technical report, November 2018.

[Pîr19] George Pîrlea. Toychain formally verified blockchain consensus. Master's thesis, University College London, 2019.

[PM15] Adam Petcher and Greg Morrisett. The foundational cryptography framework. In *POST*, 2015.

[PS18] George Pîrlea and Ilya Sergey. Mechanising blockchain consensus. In *CPP*, 2018.

[Ren19] Ling Ren. Analysis of Nakamoto consensus. ePrint, 2019.

[RVVV18] Vincent Rahli, Ivana Vukotic, Marcus Völp, and Paulo Jorge Esteves Veríssimo. Velisarios: Byzantine fault-tolerant protocols powered by Coq. In *ESOP*, 2018.

[SM19] Matthieu Sozeau and Cyprien Mangin. Equations reloaded: High-level dependently-typed functional programming and proving in Coq. In *ICFP*, 2019.

[SWT18] Ilya Sergey, James Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. *POPL*, 2018.

[Tea20] The Coq Development Team. The coq proof assistant, version 8.11.0. 2020.

[WWA+16] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. Planning for change in a formal verification of the raft consensus protocol. In *CPP*, 2016.

[WWP+15] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *PLDI*, 2015.