Verifying Gasper with Dynamic Validator Sets in Coq

Musab A. Alturki
Runtime Verification, Inc.
musab.alturki@runtimeverification.com

Elaine Li
Runtime Verification, Inc.
elaine.li@runtimeverification.com

Daejun Park
Runtime Verification, Inc.
daejun.park@runtimeverification.com

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

Karl Palmskog
KTH Royal Institute of Technology
palmskog@kth.se

Lucas Peña
University of Illinois at Urbana-Champaign
lpena7@illinois.edu

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

Abstract

Gasper is an abstract proof-of-stake protocol layer that is implemented by the Beacon Chain protocol, the underlying protocol of the Ethereum 2.0 network. A key component of Gasper is a finality mechanism that extends the original Casper FFG protocol primarily with k-finalization. In addition to the accountable safety and plausible liveness properties, Gasper describes how the protocol generalizes to the setting of dynamic validator sets and gives a lower bound on validator stake that can provably be slashed in this setting. This report describes our effort to model and verify this finality mechanism of Gasper with dynamic validator sets using the Coq proof assistant. We outline the salient details on finality in the protocol, describe previous verification efforts of Casper on which this work builds, and give an overview of the formal definitions and properties proved. The Coq source files are available at:
github.com/runtimeverification/beacon-chain-verification/tree/master/casper/coq

1 Introduction

Ethereum 2.0 [6, 20] is a major upgrade to the Ethereum platform [20] that introduces a new proof-of-stake protocol, the Beacon Chain protocol [5], with the primary goal of increasing efficiency, scalability and security of Ethereum. Participating nodes in the protocol, called validators, lock up a portion of their holdings of the underlying currency of the platform (Ether) so that they may be chosen as members of committees that propose and validate new blocks. The beacon chain protocol is built on top of important gadgets, including Casper FFG [10] for finalizing blocks and slashing finalization violations, LMD GHOST [11] as the fork-choice rule for determining the canonical chain, a rewards/penalties system for incentivizing proper validation behavior, a mechanism for managing validators joining and exiting the network, and a form of RANDAO-based pseudo-randomization for decentralized committee selection.

Gasper [11] is an abstraction of the Beacon Chain protocol that focuses on finality of blocks. A key component of Gasper is a finality mechanism that generalizes the original Casper FFG protocol, primarily by allowing a more general form of finalization. As in other Byzantine Fault Tolerant protocols, a key underlying assumption is that a super-majority of validators (by deposited stake) are honest and are following the protocol. If a validator attempts to deviate from the protocol (either intentionally to mount an attack or unintentionally due to an implementation error) and submits conflicting votes to blocks, the validator is penalized by having some significant portion of its deposited stake slashed. The protocol defines what constitutes conflicting votes, called the slashing conditions. Gasper aims to provide a high-level and mathematically precise description of finality that can be used to state and prove these conditions and the two key properties of the protocol: accountable safety and plausible liveness:

- Accountable Safety: No two blocks belonging to two different forks in the blockchain are both finalized unless at least $\frac{1}{3}$ of validators (by deposit) is provably slashable.
- Plausible Liveness: Regardless of what happened in the past, the block finalization process can never be deadlocked.

Gasper describes another generalization of Casper in which the set of validators can change over time, which is implemented by the Beacon Chain protocol. Validators may join the network (by depositing stake) or leave the network (and reclaim their staked deposit). Dynamic validator sets introduce another challenging problem: the system is now less

Project Report, June 8, 2020, Runtime Verification, Inc.
2020.

likely to be able to provably slash the misbehaving validators since they may now misbehave and then leave the network before their deposits are actually slashed. Therefore, in addition to safety and liveness, Gasper describes a third property, the slashable bound, that gives a lower bound on the slashable stake in the system expressed in terms of validator activations and exits, which can be controlled by external policies. Finally, Gasper also handles "probabilistic liveness" [11], which quantifies the likelihood of finalizing a new block given certain assumptions about good validator behavior. This property, however, is outside the scope of this work.

In this work, we formalize the finality mechanism of Gasper (the latest Casper version) in the general setting of dynamic validator sets in Coq. We state and prove all three key properties of Gasper in this setting: accountable safety, plausible liveness and the slashable bound theorem, all in the same Coq model. Deductive verification with Coq of the protocol gives the greatest confidence in the correctness and completeness of arguments, ensuring that there are no unstated assumptions or invalid deduction steps. The formalization also feeds back into the description of the protocol making it more precise and complete. We describe our effort in this report and explain how the formalization was developed and in what ways this formalization helped improve the original statements and arguments in the Gasper paper [11].

This work builds on our previous initial attempt at mechanizing the original Casper FFG protocol [2, 17, 18], which in turn benefited greatly from earlier work by others [4, 15, 19]. The four major improvements/additions compared with our previous work are:

1. Unifying the two models previously developed for safety and liveness into one model against which all properties are stated and proved. As a side product of this unification, some facts that were only assumed in our earlier models are now provable results, while other assumptions had to be slightly modified.
2. Modeling the more general k-finalization of Gasper, in which the finalization chain can be of any length $k \geq 1$.
3. Modeling the more general setting of dynamic validator sets, resulting in updated proofs of accountable safety and plausible liveness.
4. Modeling validator set weights and proving the slashable bound theorem.

The report is organized as follows. In the next section, we give some background on relevant work on formalizing Casper in Coq. This is followed in Section 3 by a description of our modeling approach. Section 4 explains the accountable safety formalization and proof. Then, in Section 5, we describe the Plausible Liveness property. We then explain in Section 6 how the slashable bound theorem is specified and proved. Section 7 discusses the results as they compare with Gasper's description. Section 8 concludes the report.

2 Background

This section explains blockchain and Casper terminology, provides pertinent Coq background, and describes the previous formalization and verification efforts we build upon.

2.1 Blockchain and Casper Terminology

Abstractly, the global state in a blockchain system is a *block forest*, with a unique genesis block that is the root of a special *block tree*. Trees not rooted in the genesis block may be possible but are typically disregarded. As new blocks arrive to the system, nodes in the system continually establish consensus on a canonical blockchain defined by one of the leaves in the special block tree. New blocks are minted through a *proposal* mechanism, which could be an underlying blockchain using proof-of-work [16] or proof-of-stake [9] to regulate block creation. Participating nodes use a local *fork choice rule* to decide where to construct a new block onto the current block tree. Due to, e.g., delays or adversarial behavior, there may be competing leaf blocks of similar tree height, defining different blockchain *forks*.

Casper, the finalization tool in Gasper, overlays a blockchain system, and intuitively works by engaging a group of autonomous *validators* who attest to, by broadcasting votes, that certain blocks in the special tree belong to the designated canonical blockchain. To participate, validators must demonstrate that they have a stake in the blockchain system by locking up a *deposit* of the blockchain's cryptocurrency. The deposit will be *slashed* if the validator is verifiably reported by other validators to be behaving adversarially. Moreover, validators may be allowed to join and leave the network and thus the set of active validators may change over time during the execution of the protocol.

For verification, we focus on the three main properties of Gasper that were proved informally in [11] (i.e. without using a mechanical proof assistant): accountable safety, plausible liveness and the slashable bound theorem, assuming dynamic validator sets. Accountable safety intuitively states that conflicting blocks in different block tree forks cannot both be finalized if more than $\frac{2}{3}$ of validators (by deposit) behave honestly. Plausible liveness states that regardless of what has happened before, it is always possible to continue to finalize blocks when more than $\frac{2}{3}$ of validators follow the protocol. Finally, the slashable bound theorem expresses a lower bound on how much stake is provably slashable (when safety is violated) in terms of activation and exit parameters that are controllable using external policies.

2.2 Casper Formalizations

Yoichi Hirai formalized and verified several earlier variants of Casper in the Isabelle/HOL proof assistant [15]. These formalizations are highly abstract, in the sense that they ignore most details of the structure of hashes, blocks, and validators. For example, the requirements on the fractions of honest validators is captured, via Isabelle's locale mechanism [8], by postulating abstract types $'q_1$ and $'q_2$ to represent collections of sets of validators of at least $\frac{2}{3}$ weight and sets of validators of at least $\frac{1}{3}$ weight respectively. Instead of any numerical details, the Isabelle formalization just assumes the key intersection property, that any two set of validators each of at least $\frac{2}{3}$ have a common subset of weight at least $\frac{1}{3}$. This is how that property was expressed in Isabelle:

$$\bigwedge q_1\ q_2\ .\ \exists\, q_3\ .\ \forall\, v\ .\ v \in_2 q_3 \longrightarrow v \in_1 q_1 \wedge v \in_1 q_2$$

Here, $\bigwedge$ is universal quantification, the subscripted operator $v \in_1 q$ means that validator v is a member of a set q which belongs to the type $'q_1$ that represents sets of weight at least $\frac{2}{3}$, and the subscripted operator $v \in_2 q$ means that validator v is a member of a set q which belongs to the type $'q_2$ that represents sets of weight at least $\frac{1}{3}$. In this proposition q_1 and q_2 have type $'q_1$ and q_3 has type q_2. This use of numbered "q"s as both types and variables is confusing, but exactly follows the Isabelle definition. While accountable safety is verified for the most recent Casper, plausible liveness is only proven for an earlier variant with different inter-validator messages. Moreover, these proofs were developed in an older version of Isabelle and already the proof of accountable safety cannot be checked with Isabelle2017.

In earlier work [2, 17, 18], we developed models in Coq of Casper based on Hirai's formalization. The models enabled mechanizing accountable safety and plausible liveness proofs of the then-updated Casper, assuming a fixed validator set. The work we present here in this report builds on our earlier developments and extends them in several ways as explained in Section 1.

2.3 Mathematical Components and Toychain

We employ several existing Coq libraries which already formalized the majority of the mathematics we need to define and reason about Casper. Mathematical Components [7] is a Coq library based on packaging mathematical structures and results in the form of Coq *canonical structures*, which can be reused and specialized when required [13]. The library was used by Gonthier et al. to capture finite group theory and prove fundamental results in abstract algebra [14]. In addition to structures from abstract algebra, the library also contains encodings of and results about many standard data structures, such as numbers, lists, and finite sets.

Toychain [4, 19] is a general formalization of blockchain systems in Coq using the Mathematical Components library. It defines blocks, forks, and distributed node state, but abstracts from specific block proposal mechanisms and procedures to let nodes decide between forks. Toychain represents a block tree as a finite map from hashes to blocks. Toychain describes the behavior of a blockchain system as a relation between global states, and establishes that absent adversarial interference, the canonical chain becomes known to all nodes in the steady state. For example, the Toychain global state is represented as a Coq record

```
Record World := mkW { localState : StateMap;
 inFlightMsgs : seq Packet; consumedMsgs : seq Packet; }.
```

where `localState` maps node names to their current block tree and other local data. We have extended and revised Toychain in collaboration with its authors to support capturing full realistic blockchain system specifications such as that for Bitcoin [1]. Our model of Casper incorporates definitions and lemmas from this extended version of Toychain.

3 Modeling and Verification Approach

As highlighted above, our approach is based on a translation of Hirai's Casper definitions and theorems [15] from Isabelle/HOL to Coq, and relating these definitions with key definitions from Toychain. While developing the model and the proofs, we have strived to stay close to the definitions and arguments in the Gasper paper [11] as much as possible, which helps readability of the model, and makes it easier to verify correctness of the arguments and identify any missing assumptions or reasoning steps. Naturally, the model is much more rigorous and there are aspects of the model that are necessarily different. We highlight these aspects of the model later in Section 7.

We used concepts from the Mathematical Components library to simplify forming and reasoning about finite sets, finite maps and natural numbers and expressions (see [7]). Since the formal model for establishing safety by Hirai mostly uses first-order reasoning, we were able to successfully leverage the CoqHammer extension [3, 12] to perform safety proofs in Coq that closely followed Isabelle/HOL proofs. The project's Github repository at github.com/runtimeverification/beacon-chain-verification/tree/master/casper/coq lists the project's dependencies needed to check the proofs.

Below, we highlight the main components of the model and how they are specified in Coq.

3.1 Validators and Weights

Validators are participating nodes in the network that contribute to the protocol by validating blocks, casting votes attesting to blocks and proposing new blocks. Validators are identified by fixed-size keys so we represent validators as members of a *finite type* (having a finite number of members that can be enumerated), written `Validator : finType`.

For a node to become a validator, it needs to have an amount of currency (Ether) locked up as its stake in the network. We represent this fact by an abstractly declared finite map (a function whose domain is a finite type) that gives the stake amount of a validator `stake : {fmap Validator →nat}`. Every validator must have a stake in the system (although the stake may in principle reach zero due to slashing). Therefore, we add the assumption that the stake map is total:

```
Axiom st_fun : ∀ v : Validator, v ∈ stake.
```

Moreover, we will need to reason about the combined deposits, or *weights*, of validator sets, defined as the total stake amount of all validators in a set:

```
Definition wt (s:{set Validator}) : nat :=
  \sum_(v in s) stake.[st_fun v].
```

Given a set of validators `s`, the function `wt` computes the weight of `s` as the sum of stake amounts of all validators in that set. The expression `stake.[st_fun v]` is the stake for validator `v` (which we know is well-defined since `stake` is assumed total — the term `st_fun v` is a proof that `v` is in the domain of `stake`).

Several important properties of `wt` follow from its definition and basic set-theoretic laws. For example, the weight function is monotonic:

```
Lemma wt_inc_leq (s1 s2:{set Validator}):
  s1 \subset s2 → wt s1 <= wt s2.
```

Another example is that the weight of two disjoint validator sets is exactly the sum of their weights (the symbol `:|:` denotes set union):

```
Lemma wt_join_disjoint (s1 s2:{set Validator}):
  [disjoint s1 & s2] → wt (s1 :|: s2) = wt s1 + wt s2.
```

These properties and several others along with their proofs can be found in `Weights.v` in the project's repository.

3.2 Block Forests

The finality mechanism of Gasper (and Casper in particular) operates on epoch boundary blocks, also known as *checkpoint blocks*. We therefore consider only checkpoint blocks in the model, and refer to them simply as blocks. A block is identified by a (fixed-width) hash value, so we assume a finite type of block hashes `Hash : finType` (with member `genesis` representing the Genesis block). To abstract a block forest, we assume a binary parent relationship on block hashes `hash_parent : rel Hash`, for which we use the notation `h1 <~ h2` to mean that `h1` is a (the) parent of `h2`, and having the following two properties:s

1. A block cannot be the parent of itself:

   ```
   Axiom hash_parent_irreflexive:
     ∀ h1 h2, h1 <~ h2 → h1 ≠ h2.
   ```

2. A block can have at most one parent block:

   ```
   Axiom hash_at_most_one_parent :
     ∀ h1 h2 h3, h2 <~ h1 → h3 <~ h1 → h2 = h3.
   ```

We also define the "ancestor" binary relation `<~*` as the reflexive-transitive closure of `<~`, using the closure operation `connect`:

```
Definition hash_ancestor h1 h2 := connect hash_parent h1 h2.
```

The notion of block ancestry is fundamental to the protocol as it enables specifying conflicting blocks, which are blocks that are not related by the ancestry relation. Several properties of the ancestry relation follow from those of the closure operation `connect`, such as reflexivity (every block is an ancestor of itself) and concatenation (an ancestor of an ancestor is also an ancestor) among others. One basic property with special significance in the accountable safety proof is that a conflicting block cannot belong to the ancestry of that block:

```
Lemma hash_ancestor_conflict:
  ∀ h1 h2 p, h1 <~* h2 → p </~* h2 → p </~* h1.
```

The notation `</~*` expands to the negation of the ancestry relation `<~*`.

Finally, we give an inductive definition of ancestry that makes the exact number of steps explicit:

```
Inductive nth_ancestor : nat → Hash → Hash → Prop :=
| nth_ancestor_0 : ∀ h1, nth_ancestor 0 h1 h1
| nth_ancestor_nth : ∀ n h1 h2 h3,
    nth_ancestor n h1 h2 → h2 <~ h3 →
    nth_ancestor n.+1 h1 h3.
```

We show that the two definitions are compatible in the sense that they both capture the same notion of ancestry. The additional piece of information provided by `nth_ancestor` will be useful when defining justification and finalization.

3.3 The Global State

Given our definitions of validators and the block forest, we define the global state as the set of votes cast by validators. Similarly to how votes are defined in Casper, we model a vote by a quintiple:

```
Definition Vote := (Validator * Hash * Hash * nat * nat)%type.
```

Each vote `(v, s, t, s_h, t_h)` is signed by a particular validator (attester) `v`, and names the source and target blocks (`s` and `t` respectively) and their attestation heights (`s_h` and `t_h` respectively).

The global state is defined by the finite set of votes cast:

```
Definition State := {fset Vote}.
```

Whether a vote has been cast or not can be determined by a boolean membership predicate:

```
Definition vote_msg (st:State) v s t s_h t_h : bool :=
  (v,s,t,s_h,t_h) ∈ st .
```

3.4 Slashing Violations

With votes defined we can then define the two slashing conditions of Casper (shown in Figure 1) in terms of two kinds of conflicting voting behaviors: double-voting and surround-voting. Double-voting occurs if validator `v` has

An individual validator v must not publish two distinct votes,

$$\langle v, s_1, t_1, h(s_1), h(t_1)\rangle \quad \text{and} \quad \langle v, s_2, t_2, h(s_2), h(t_2)\rangle,$$

such that either:

I. $h(t_1) = h(t_2)$. Equivalently, a validator must not publish two distinct votes for the same target height.

II. $h(s_1) < h(s_2) < h(t_2) < h(t_1)$. Equivalently, a validator must not vote within the span of its other votes.

Figure 1. Slashing Conditions of Casper

voted for two different target blocks `t1` and `t2` both at height `t_h`.

```
Definition slashed_double_vote st v :=
  ∃ t1 t2, t1 ≠ t2 ∧ ∃ s1 s1_h s2 s2_h t_h,
    vote_msg st v s1 t1 s1_h t_h
    ∧ vote_msg st v s2 t2 s2_h t_h.
```

Surround-voting happens if validator `v` has made two votes so the source height `s1_h` and target height `t1_h` of one vote strictly surround the source height `s2_h` to target height `t2_h` range of the other vote.

```
Definition slashed_surround_vote st v :=
  ∃ s1 t1 s1_h t1_h,
  ∃ s2 t2 s2_h t2_h,
    vote_msg st v s1 t1 s1_h t1_h ∧
     vote_msg st v s2 t2 s2_h t2_h ∧
     s2_h > s1_h ∧ t2_h < t1_h.
```

We now define a proposition saying a validator is slashed in a given global state.

```
Definition slashed st v : Prop :=
 slashed_double_vote st v ∨ slashed_surround_vote st v.
```

A validator is slashed if it has either double- or surround-voted.

3.5 Quorums

As is typical in Byzantine fault tolerant protocols, Gasper assumes that the supermajority of validators are honest. A supermajority is defined as a subset of validators whose weight is at least $\frac{2}{3}$ of the total stake. Moreover, proofs of safety and liveness generally show that an attack cannot succeed without a set of attacking validators of at least $\frac{1}{3}$ of the total stake being slashed for it. As we intend to model dynamic validator sets where the active set of validators may change across different blocks, these bounds on weights can only be defined in relation to the current set of active validators.

Therefore, we first declare an abstract (finite) map vset : `{fmap Hash →{set Validator}}` that gives the active validator set for a block. Every block has a set of active validators, so we declare the map total:

```
Axiom vs_fun : ∀ h : Hash, h ∈ vset.
```

Now the $\frac{2}{3}$ lower bound on weights of validator sets (quorums) can be specified as a predicate parametric to blocks:

```
Definition quorum_2 (vs : {set Validator}) (b : Hash) : bool :=
  (vs \subset vset.[vs_fun b]) &&
  (wt vs >= two_third (wt vset.[vs_fun b])).
```

For a block `b`, a validator set `vs` is a `quorum_2` set if it satisfies two conditions: (1) all validators in `vs` appear in the active validator set of the block `b`, and (2) the weight of the set `vs` is at least $\frac{2}{3}$ of the total weight of all active validators of `b`. (note that the constant `two_third` is merely used for convenience of typing with natural numbers).

The predicate `quorum_2` abstracts the powerset of sets of validators with weight at least $\frac{2}{3}$ of the total for a given block. For example, justifying a link from a source block to a target block requires receiving votes from all validators in a supermajority set (with respect to the target block), i.e. a subset of validators satisfying `quorum_2` for the target block.

Given these definitions, we can now specify what it means for a quorum to be slashed using abstract membership constraints:

```
Definition q_intersection_slashed st :=
  ∃ (bL bR: Hash) (vL vR: {set Validator}),
    vL \subset vset.[vs_fun bL] ∧
    vR \subset vset.[vs_fun bR] ∧
    quorum_2 vL bL ∧
    quorum_2 vR bR ∧
    ∀ v, v ∈ vL → v ∈ vR → slashed st v.
```

The proposition `q_intersection_slashed` states that slashing a quorum means the existence of two supermajority quorums `vL` and `vR` with respect to some blocks `bL` and `bR` respectively whose intersection is slashed. Note that unlike Hirai's abstract formulations of Casper [15] where the intersection is guaranteed to be of weight at least $\frac{1}{3}$ of the total stake, the formulation in Gasper is more general since the validator sets are dynamic and the intersection is no longer guaranteed to respect this bound (more on this in Section 6).

3.6 Justification

A supermajority link from a source block to a target block is formed when a supermajority quorum of validators (with respect to the target block) vote for this pair of blocks. We capture this intuition by first defining the set of validators attesting for a given pair of blocks (using set comprehension notation):

```
Definition link_supporters st s t s_h t_h : {set Validator} :=
  [set v | vote_msg st v s t s_h t_h ].
```

Now we can define a supermajority link predicate as follows:

```
Definition supermajority_link (st:State) s t s_h t_h : bool :=
  quorum_2 (link_supporters st s t s_h t_h) t.
```

We have, in the current state, a supermajority link from block `s` to `t` with justification heights `s_h` and `t_h` respectively if and only if the set of validators voting for this link is a

supermajority quorum (relative to the active validator set of the target block `t`).

Note that a supermajority link as defined above does not yet capture proper justification links in Gasper, which must be valid forward links in the block forest with the target block being a descendant block of the source at the appropriate heights:

```
Definition justification_link (st:State) s t s_h t_h : Prop :=
  t_h > s_h ∧
  nth_ancestor (t_h - s_h) s t ∧
  supermajority_link st s t s_h t_h.
```

The conjuncts respectively require that the justification height of the target block `t_h` is strictly greater than that of the source, that the source block `s` is actually the ancestor the appropriate number of levels above the target block `t`, and that a supermajority quorum of validators have voted for this link in the state.

This allows us to define justified blocks inductively in terms of a path from the genesis block all the way to that block:

```
Inductive justified (st:State) : Hash → nat → Prop :=
| justified_genesis : justified st genesis 0
| justified_link : ∀ s s_h t t_h,
    justified st s s_h →
    justification_link st s t s_h t_h →
    justified st t t_h.
```

The genesis block is always justified at height 0. A non-genesis block is justified if there exists a proper justification link to the block from an ancestor block that is already justified.

3.7 Finalization

A justified block is finalized if there exists an immediate descendant (child) block that is also justified by a supermajority link to it. This is the original definition of finalization in Casper [10], which is captured by the following definition:

```
Definition finalized st b b_h :=
  justified st b b_h ∧
  ∃ c, (b <~ c ∧ supermajority_link st b c b_h b_h.+1).
```

Gasper [11] generalizes finalization to k-finalization: a k-finalized block is a justified block that has a k-descendent who is also justified by a supermajority link to the block, and all blocks to the descendent are also justified. We formalize this as follows:

```
Definition k_finalized st b b_h k :=
  k >= 1 ∧
  ∃ ls, size ls = k.+1 ∧
       head b ls = b ∧
       (∀ n, n <= k →
            justified st (nth b ls n) (b_h+n) ∧
            nth_ancestor n b (nth b ls n)
       ) ∧
       supermajority_link st b (last b ls) b_h (b_h+k).
```

The chain `ls`, which is of length `k + 1`, has the k-finalized block `b` as its head. Each block in the chain is justified, with the last block (the k-descendent of `b`) justified by a supermajority link from `b`.

Note that $k \geq 1$. When `k` is 1, the two notions of finalization and k-finalization coincide:

```
Lemma finalized_means_one_finalized : ∀ st b b_h,
  finalized st b b_h ↔ k_finalized st b b_h 1.
```

4 Accountable Safety

This section describes the Coq formalization and verification of Casper's *accountable safety* property. A major part of the Gasper (Casper) design is ensuring that an attack cannot finalize blocks on both sides of a fork in the chain without the attacker losing a significant amount of money by having their validator deposits slashed. This is formalized as the accountable safety theorem. In the fixed validator sets setting, the theorem says that if two blocks are finalized and neither is an ancestor of the other, then validators having at least $\frac{1}{3}$ of the total stake must have violated the slashing conditions [10]. In the more general setting where validators are allowed to join and leave the network, the lower bound on the deposits that can be slashed is dependent on how much validator churn is allowed, which can be controlled by externally specified validator activation and exit policies [11]. Note that the difference between slashing violations and actually losing funds is addressed and handled with other countermeasures in the protocol [5, 10].

We model the dynamic validator setting, of which the fixed validator sets setup can be obtained as a special case. We also capture Gasper's generalized accountable safety theorem in which each of the two conflicting blocks can be k-finalized for potentially two distinct values of k.

Building on the definitions of validator quorums and justification described above, we define a k-finalization fork in a state as two conflicting k-finalized blocks (hashes):

```
Definition k_finalization_fork st k1 k2 :=
  ∃ b1 b1_h b2 b2_h,
    k_finalized st b1 b1_h k1 ∧
    k_finalized st b2 b2_h k2 ∧
    b2 </~* b1 ∧ b1 </~* b2.
```

Note that the blocks are finalized at depths `k1` and `k2`, which may be different. Forking with same-depth finalization or with finalization at depth exactly 1 are all instances of this definition.

With `q_intersection_slashed st` (defined in Section 3.5) capturing that a quorum of validators is slashed in state `st` due to a violation of the slashing conditions in Figure 1, we define and prove accountable safety:

```
Theorem k_accountable_safety : ∀ st k1 k2,
  k_finalization_fork st k1 k2 → q_intersection_slashed st.
```

The proof roughly follows the informal arguments [11]:

1. If the two finalized blocks `b1` and `b2` were at the same height, then the validator sets corresponding to these blocks have a common quorum of validators who violated slashing condition I (double-voting for two different target blocks with the same height).
2. Otherwise, consider the path of justified links which justifies the higher finalized block (say that the higher finalized block is b2 — the other case is symmetrical). If any justified block along this path has the same height as the other justified block `b1` or any of its justified descendants along its k1-finalization chain, then we again have a set of validators violating slashing condition I.
3. Otherwise, find the justified link along that path with a source lower than the other justified block `b1` and a target higher than any of its descendants belonging to its k1-finalization chain. Taking this link along with the justified link between the k1-finalized block `b1` and its k1-descendent, we have a set of validators violating slashing condition II.

The mechanization in Coq structures the proofs so that each of three main cases above is proved as a separate result. Note that the third case above uses induction to traverse the path of justification links in the argument. In all cases, the proofs find exactly the pair of blocks whose validator sets have an intersection that is slashed for violating the conditions, as required to conclude `q_intersection_slashed`.

5 Plausible Liveness

The liveness goal for Casper is that as long as a $\frac{2}{3}$ quorum of validators (by weight) always exists and is following the protocol and block proposals continue, then further checkpoints can continue to be finalized regardless of the behavior of the rest of possibly misbehaving validators, without needing any of the honest validators to violate a slashing condition and sacrifice their deposit to allow the chain to live.

Our Coq proof is based on the argument given in [10, 11], and also Yoichi Hirai's Isabelle/HOL proof of plausible liveness for an older variant of the Casper protocol [15]. In our mechanization, we state and prove liveness in terms of the hash and height of the most recent previously-finalized block (which we prove always exists). The proof uses the same model of checkpoint blocks described in Section 3 and builds on the accountable safety result described in Section 4 above. Therefore, compared with previous mechanizations efforts, our mechanization in Coq proves Gasper's plausible liveness property in a more general setting with dynamic validators and a generalized finalization definition.

5.1 Proof Strategy and Assumptions

To formalize a statement of plausible liveness, we abstract away from any implementation details of "following the protocol", and simply require that our $\frac{2}{3}$ good validators have not made certain sorts of bad votes. For the conclusion we also ignore the details of how validators decide to make votes, and simply show that there is some set of votes which the good validators could make that would finalize a further block and not violate slashing conditions.

In Hirai's proof for the previous Casper design it was sufficient to simply require that the good validators were unslashed. The current Casper design intentionally removed any slashing conditions that required knowing the state of the chain when the votes were made. We found that *one of these conditions was essential to the proof*, so our definition of good behavior for the current Casper protocol requires that a validator has not made votes with unjustified sources or votes that would represent invalid backward justification links, in addition to requiring that a good validator had not been slashed.

Therefore, we define the property that, for any block, a super-majority-quorum validator makes only good votes:

```
Definition good_votes (st : State) :=
  ∀ b q2, quorum_2 q2 b →
    ∀ v, v ∈ q2 →
    justified_source_votes st v ∧ forward_link_votes st v.
```

The proposition `justified_source_votes` holds when all votes cast by validator `v` in state `st` have sources that are justified, while the proposition `forward_link_votes` is satisfied when each vote by v in st links a source block to a descendant target block at the appropriate heights.

We also define the condition that, for any block, a super-majority of validators exists and is good given a set of votes:

```
Definition two_thirds_good (st : State) :=
  ∀ b, ∃ q2, quorum_2 q2 b ∧
    ∀ v, v ∈ q2 → ~ slashed st v.
```

Recall that `slashed` is the property that a validator committed a slashing violation in the given state.

Another explicit hypothesis of our plausible liveness theorem captures the assumption that block proposals continue. We only need to assume that blocks can be found above the highest justified block, which we prove exists and is unique based on accountable safety, but we may need to find a block at arbitrary height. This definition captures the property of being the highest justified block:

```
Definition highest_justified st b b_h : Prop :=
  ∀ b' b_h', b_h' >= b_h
  → justified st b' b_h'
  → b' = b ∧ b_h' = b_h.
```

The following defines the property of having descendants at arbitrary heights:

```
Definition blocks_exist_high_over (base : Hash) : Prop :=
  ∀ n, ∃ block, nth_ancestor n base block ∧ n > 1.
```

Note that this property ensures that block production continues so that a descendent chain of `base` of length at least 2 (the condition `n > 1`) exists. Finalization (for non-genesis blocks)

needs at least a chain of length 2 (for 1-finalization) to form, and we are not making any assumption on the justification status of `base`.

One important result (assumed as a hypothesis in [2]) is that, assuming good behavior of super-majority quorums, there always exists a justified block that is the highest justified for any set of votes:

```
Lemma highest_∃: ∀ st,
  ~ q_intersection_slashed st →
  good_votes st →
  ∃ b b_h,
    justified st b b_h ∧
    highest_justified st b b_h.
```

If the state `st` has no justification links, then the genesis block is the highest justified block. Otherwise, a justification link exists, and the assumption of good votes (and the fact that `st` is finite) guarantees existence of a maximal justification link whose target is at least as high as any other link. By the assumption that no super-majority quorum violated slashing, and the accountable safety theorem, the maximal link target exists and is unique.

5.2 Theorem

With these ingredients, this is the statement of plausible liveness that we prove:

```
Theorem plausible_liveness : ∀ st,
  two_thirds_good st →
  ~ q_intersection_slashed st →
  good_votes st →
  (∀ b b_h, highest_justified st b b_h → blocks_exist_high_over b) →
  ∃ st', unslashed_can_extend st st'
    ∧ no_new_slashed st st'
    ∧ ∃ (new_finalized new_final_child:Hash) new_height,
        justified st' new_finalized new_height
        ∧ new_finalized <~ new_final_child
        ∧ supermajority_link st' new_finalized new_final_child
                             new_height new_height.+1.
```

The desired conclusion of the theorem is expressed by stating that there exists a set of votes `st'`, such that any votes which were not already in `st` were made by unslashed validators, that no validators that were unslashed in `st` are slashed in `st'`, and that there is a block `new_finalized` which is finalized under votes `st'` and is higher than the previous finalized block.

The proof proceeds as outlined in [10, 11]: We take the highest justified checkpoint J which is a descendant of the most recently finalized block, consider the maximum height of the target of any existing vote, and use the assumption that block proposals continue to find a descendant A of J at a greater height than that, and an immediate child B of A. Then all good validators will vote for the links J⟶A and A⟶B. This is easily shown to finalize A and require votes only from good validators. It remains only to prove that this does not violate any slashing condition. Recall the definitions of the slashing conditions from Figure 1.

- Condition I is not violated because a good validator's two new votes have targets at different heights, and any previous votes have a target at a lower height than A, by choice of A.
- Condition II is not violated because the two new votes of good validators do not nest, a previous vote cannot have a range surrounding a new vote because all previous votes have targets below A by choice of A, and a previous vote cannot nest within the vote for J⟶A because it would have a source above J, we assume existing votes from good validators have justified sources, and J is the highest justified block.

5.3 Unslashed is not enough

One might wonder if a different voting strategy could prove plausible liveness while only needing to require that the good validators be unslashed. To show that there cannot be such a proof, we give an example where all validators are unslashed but it is impossible to justify any further blocks (let alone finalize a further block). Every validator in this example has made one vote with an unjustified source.

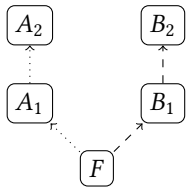

Figure 2. Unslashed Progress Counterexample

The example is shown in Figure 2. F is the most recent finalized block, A_1 and B_1 immediate children of F, and A_2 and B_2 immediate children (respectively) of A_1 and A_2. Half of the validators have voted for the edges $F \longrightarrow A_1$ and $A_1 \longrightarrow A_2$ (dotted arrows) and the other half of the validators have voted for $F \longrightarrow B_1$ and $B_1 \longrightarrow B_2$ (dashed arrows). It is impossible to make a super-majority link from F to any other block without some validators being slashed. A new edge from F to a block one or two levels above F would violate slashing Condition I, because every validator already has votes with a target at that level. An edge from F to a block at any higher level would violate Condition II, with the inner vote being that validator's vote for $A_1 \longrightarrow A_2$ or $B_1 \longrightarrow B_2$.

6 Slashing Lower Bound

When the set of validators is fixed, i.e. validators are not allowed to join or leave the network during the execution of the protocol, being a super-majority quorum depends only on the condition that the quorum's weight is at least $\frac{2}{3}$ of the total stake. Consequently, the intersection of two super-majority quorums in this setting must have at least $\frac{1}{3}$ of the

total stake, and hence the lower bound of $\frac{1}{3}$ of the total stake being slashed in case of a violation of safety.

However, requiring that the set of validators be fixed is unrealistic, since the Beacon Chain protocol is a permissionless system, allowing nodes to join and leave (according to some activation and exit policies). Intuitively, nodes who have their deposits accepted may be activated at a block and remain active until they decide to leave or are forced to leave (when its deposited stake becomes too low due to repeated slashing for example), at which point the validator exits the active validator set. Having dynamic sets, however, means that, in case of a safety violation, we are less likely to be able to provably slash the misbehaving validators since they may now manage to "escape" before their deposits are actually slashed. Furthermore, active validator sets at two different branches of the block tree may be so much different that finding votes violating the conditions is a problem. The Beacon Chain protocol implements a set of elaborate activation and exit policies whose primary objective is to alleviate these potential problems (see [5] for more details).

Our formalization of the protocol in Coq is guided by the level of abstraction given by Gasper [11]. As described in Section 3.5, the function `vset` models the behavior of potentially changing validator sets by defining for each block a set of active validators. Recall the quorum intersection property `q_intersection_slashed` that defines what it means to be "slash-able" in this context: there exists two super-majority quorums, with each quorum being a subset of a set of active validators of some block, such that their intersection is slashed (note how this property specializes to the $\frac{1}{3}$ bound if the set of validators is assumed fixed, i.e. when `vset` is constant). Gasper expresses a lower bound on what can be provably slashed in this setting in terms of the churn of validator sets in the checkpoint block tree, which is practically useful as this churn can be controlled through validator activation and exit policies, effectively providing a mechanism to adjust the acceptable bound on what is guaranteed to be slashable.

To formalize this theorem, we first define activations and exits with respect to two validator sets.

```
Definition activated (s1 s2: {set Validator}): {set Validator} :=
  s2 :\: s1.
Definition exited (s1 s2: {set Validator}): {set Validator} :=
  s1 :\: s2.
```

The symbol `:\:` is for set difference. We also define activation and exit weights:

```
Definition actwt (s1 s2: {set Validator}): nat :=
  wt (activated s1 s2).
Definition extwt (s1 s2: {set Validator}): nat :=
  wt (exited s1 s2).
```

We now have all the ingredients needed to formalize this theorem giving a lower bound on the amount of stake that can be provably slashed due to a violation:

```
Theorem slashable_bound : ∀ st b0 b1 b2 b1_h b2_h k1 k2,
  k_finalized st b1 b1_h k1 →
  k_finalized st b2 b2_h k2 →
  b1 </~* b2 → b2 </~* b1 →
  ∃ (bL bR:Hash) (qL qR:{set Validator}),
    let v0 := vset.[vs_fun b0] in
    let vL := vset.[vs_fun bL] in
    let vR := vset.[vs_fun bR] in
    let aL := actwt v0 vL in
    let eL := extwt v0 vL in
    let aR := actwt v0 vR in
    let eR := extwt v0 vR in
    let xM := maxn (wt vL - aL - eR) (wt vR - aR - eL) in
      qL \subset vL ∧
      qR \subset vR ∧
      wt (qL :&: qR) >= xM - one_third (wt vL) - one_third (wt vR).
```

The theorem states that if two conflicting blocks `b1` and `b2` are finalized, which is the premise of a safety violation, then we can find two quorums `qL` (for some block `bL`) and `qR` (for some block `bR`) such that the weight of the intersection `(qL :&: qR)` is at least the quantity:

```
maxn (wt vL - aL - eR) (wt vR - aR - eL)
- one_third (wt vL)
- one_third (wt vR)
```

where `maxn` evaluates to the maximum of its two arguments. Note that this quantity is parameterized by the activation and exit weights (`aL`, `aR`, `eL` and `eR`) with respect to a reference block `b0`. The intersection of the two quorums is the set of validators that have violated the slashing conditions and are thus to be slashed.

The proof follows the argument given in [11], which is based on some basic properties of the `wt` function (some of which were highlighted in Section 3.1 above). The proof also uses a number of basic set-theoretic lemmas and several inequalities on natural numbers from the Mathematical Components library, augmented with a few more lemmas in the files `SetTheoryProps.v` and `NatExt.v` in the project's repository.

7 Discussion

As we mentioned above, our strategy in developing the formalization in Coq is to have a model that matches as much as possible the level of abstraction of Gasper as described in [11]. The various representations in the model, e.g. the checkpoint block tree and validator sets and votes, along with the definitions and properties, such as those for justification, finalization and the theorem statements, are all based directly on the protocol's description. Nevertheless, one deviation from the description was made to avoid unnecessarily complicating the model. More specifically, our model uses the original Casper paper's abstract treatment of checkpoint blocks [10], which does not explicitly use justified pairs as in Gasper [11]. Qualifying a justified block with the justification epoch is not needed in our model because it already abstracts blocks by their identifiers (hashes), which

are assumed unique. So even if the same block is justified for two different epochs, the model already treats these two justification instances as two distinct blocks. The additional information that the two instances actually point to the same block is irrelevant to the model.

Furthermore, the formalization is naturally much more rigorous than a high-level description and makes explicit all the assumptions needed for the arguments to hold. We highlight below some of the important assumptions that the model explicitly specifies for the plausible liveness property:

- Supermajority quorums are non-empty (`quorum_2_nonempty`), which implies the weaker condition that validator sets are non-empty. No progress can be made with an empty quorum (or validator set).
- Justification links must be proper: In addition to being a supermajority link, a justification link must be a valid forward link in the block tree with the target's height being greater than the source's.
- Only the votes that originate from the validator set of the target block being voted for are considered (`votes_from_target_vset`). This also implies Justification is preserved when the state is expanded with new votes (coming from validators in the validator set of the justified block). Although this is not explicitly stated in Gasper, the Beacon Chain protocol's implementation already ignores votes that do not satisfy this assumption.
- For any block, all validators of a supermajority set of that block produce only good votes: votes that have justified sources and that represent valid forward links in the block tree (`good_votes`). Votes with unjustified sources or ones that do not represent proper forward links in the checkpoint tree do not violate any slashing conditions themselves, but can prevent a validator from contributing to progress, because votes spanning over such bad votes would violate slashing condition II. This is also checked for in the implementation of the protocol.
- For any block, there exists a supermajority whose validators have not been slashed (`two_thirds_good`). Progress is meaningful only if it can be made by validators following the protocol.

Finally, going through this exercise of modeling and verification helps in refining the definitions and arguments of Gasper. For instance, the Coq model does not assume that the genesis block is finalized, but it can be proved that this property follows from the model, under certain good-behavior assumptions that we already make for safety and liveness. This suggests a possible simplification to the definition of finalization, provided that these assumptions are given clearly. Another example is the formalization of the slashable bound theorem, which suggested improvements to the statement and argument of the theorem in Gasper's description and eventually the description was updated to incorporate them. In particular, the assumption that the reference block `B0` is finalized was shown unnecessary, and in fact this block can potentially be any block in the block tree. Furthermore, the statement of the theorem and how its argument uses Lemma 8.3 were made more accurate, matching the formalization in the model.

8 Conclusion

We presented a formalization of the finality mechanism of Gasper (most up-to-date Casper version) in Coq, along with proofs of three major properties: accountable safety, plausible liveness and the slashing lower bound, all in one model supporting the general dynamic validator sets setting. These proofs clarified the assumptions needed for the properties stated in the Gasper paper [11], and helped further improve its theorem statements and arguments. Our formalization work paves the way for proving accountable safety, plausible liveness and the slashable bound for the state-transition function of the Beacon Chain protocol's implementation [5].

Acknowledgments

We thank Danny Ryan, Carl Beekhuizen, Martin Lundfall, Yan Zhang and Aditya Asgaonkar from the Ethereum Foundation for their valuable feedback and comments. This work was funded by the Ethereum Foundation.

References

[1] 2018. Bitoychain. https://github.com/palmskog/bitoychain
[2] 2018. Casper Proofs. https://github.com/runtimeverification/casper-proofs
[3] 2018. CoqHammer. https://github.com/lukaszcz/coqhammer
[4] 2018. Toychain. https://github.com/certichain/toychain
[5] 2020. Beacon Chain. https://github.com/ethereum/eth2.0-specs/blob/dev/specs/phase0/beacon-chain.md
[6] 2020. Ethereum 2.0 Specifications. https://github.com/ethereum/eth2.0-specs
[7] 2020. Mathematical Components Project. https://math-comp.github.io/math-comp/
[8] Clemens Ballarin. 2014. Locales: A Module System for Mathematical Theories. *Journal of Automated Reasoning* 52, 2 (01 Feb 2014), 123–153.
[9] Iddo Bentov, Ariel Gabizon, and Alex Mizrahi. 2016. Cryptocurrencies Without Proof of Work. In *Financial Cryptography and Data Security*, Jeremy Clark, Sarah Meiklejohn, Peter Y.A. Ryan, Dan Wallach, Michael Brenner, and Kurt Rohloff (Eds.). Springer, Berlin, Heidelberg, 142–157.
[10] Vitalik Buterin and Virgil Griffith. 2017. Casper the Friendly Finality Gadget. *CoRR* abs/1710.09437 (2017).
[11] Vitalik Buterin, Diego Hernandez, Thor Kamphefner, Khiem Pham, Zhi Qiao, Danny Ryan, Juhyeok Sin, Ying Wang, and Yan X Zhang. 2020. Combining GHOST and Casper. arXiv:cs.CR/2003.03052
[12] Łukasz Czajka and Cezary Kaliszyk. 2018. Hammer for Coq: Automation for Dependent Type Theory. *Journal of Automated Reasoning* 61, 1 (2018), 423–453.
[13] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. 2009. Packaging Mathematical Structures. In *Theorem Proving in Higher Order Logics*. 327–342.

[14] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. 2013. A Machine-Checked Proof of the Odd Order Theorem. In *Interactive Theorem Proving*. 163–179.

[15] Yoichi Hirai. 2018. A repository for PoS related formal methods. https://github.com/palmskog/pos

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

[17] Karl Palmskog, Milos Gligoric, Lucas Pena, Brandon Moore, and Grigore Roşu. 2018. *Verification of Casper in the Coq Proof Assistant*. Technical Report. University of Illinois at Urbana-Champaign, http://hdl.handle.net/2142/102075.

[18] Karl Palmskog, Milos Gligoric, Lucas Pena, and Grigore Roşu. 2019. Verifying Finality for Blockchain Systems. In *The Fifth International Workshop on Coq for Programming Languages (CoqPL'19)*.

[19] George Pîrlea and Ilya Sergey. 2018. Mechanising blockchain consensus. In *Certified Programs and Proofs*. 78–90.

[20] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. (2014). http://gavwood.com/paper.pdf