Compositional Verification of Composite Byzantine Protocols

Qiyuan Zhao
National University of Singapore
Singapore
qiyuanz@comp.nus.edu.sg

George Pîrlea
National University of Singapore
Singapore
gpirlea@comp.nus.edu.sg

Karolina Grzeszkiewicz
Yale-NUS College
Singapore
karolina.grzeszkiewicz@u.yale-nus.edu.sg

Seth Gilbert
National University of Singapore
Singapore
seth.gilbert@comp.nus.edu.sg

Ilya Sergey
National University of Singapore
Singapore
ilya@nus.edu.sg

Abstract

Byzantine Fault-Tolerant (BFT) protocols are known to be difficult to design and to reason about. To address this challenge, on one hand, several approaches have been developed recently for computer-aided formal verification of the desired correctness properties, both safety and liveness, of standalone BFT protocols. On the other hand, the distributed computing community has made attempts to reduce the conceptual complexity of constructing new such protocols by showing how to assemble them from simpler "building blocks". No methodology to date combines these two approaches for foundational verification of arbitrary BFT protocols.

We present Bythos, the first foundational framework for compositional mechanised verification of both safety and liveness of composite BFT protocols. Bythos is implemented on top of the Coq proof assistant and uses Coq's higher-order logic to reuse proofs of common facts about knowledge and trust in BFT protocols. It allows for compact liveness specifications in the style of TLA+, and for their proofs using an embedding of TLA into Coq. Most importantly, Bythos provides a family of higher-order definitions that allow building composite BFT protocols from simpler ones, with their correctness proofs derived. We showcase Bythos by verifying in it safety and liveness properties of three basic BFT protocols: Reliable Broadcast, Provable Broadcast, and the recently proposed Accountable Byzantine Confirmer, as well as their compositions.

CCS Concepts

• **Networks** → **Protocol testing and verification**; **Formal specifications**; • **Security and privacy** → *Distributed systems security*.

Keywords

Byzantine fault tolerance, distributed protocols, formal verification

ACM Reference Format:
Qiyuan Zhao, George Pîrlea, Karolina Grzeszkiewicz, Seth Gilbert, and Ilya Sergey. 2024. Compositional Verification of Composite Byzantine Protocols. In *Proceedings of the 31st ACM Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA*. ACM, New York, NY, USA, 15 pages.

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

1 Introduction

In the evolving landscape of distributed computing, *Byzantine Fault Tolerance* (BFT) protocols have emerged as a cornerstone for ensuring the reliability and integrity of Internet services in the presence of malicious adversaries and arbitrary faults [37]. In particular, BFT protocols have proven invaluable in critical infrastructure systems and blockchain technologies [6, 16, 19, 43, 65].

After more than two decades of research on BFT protocols, computer scientists' understanding of them is relatively mature. Nonetheless, new proposals of such protocols are still intricate and implementation is difficult and prone to bugs [49], which, given the critical applications of the technology, could prove disastrous.

One approach to reducing the risk of bugs in BFT protocols is computer-aided *formal verification*. Protocols and their implementations can be encoded in the language of a software tool such as a proof assistant and shown from first principles to satisfy their desired properties [26, 30, 39, 54, 57, 61–63].

These proofs can often be made fully *foundational* [8], in the sense that they 'are 'expressed in terms of a small set of accepted axioms, and machine-checked, providing a very high degree of confidence that the verified systems indeed satisfy the properties.

Practically speaking, however, most of these efforts cannot be reused. The issue is that BFT protocols, at least for a long time, have generally been described monolithically, without a clear separation into sub-protocols with logically distinct roles of each such protocol and its effect on the overall correctness properties of the whole construction. The issue is not with formal verification itself, but with how the protocols are described in the first place, which makes it difficult to recognise reusable design components that could be used for constructing new protocols and proofs about them.

Motivating example: verifying iterated provable broadcast. Protocol designers have recognised the issue that BFT protocols are complicated to understand and have sought to provide simpler, more modular descriptions. An example of this is a series of blog posts by Abraham *et al.* [1–3] explaining how a basic Provable Broadcast (PB) protocol [55] can be *iterated* to obtain progressively stronger properties. In a single instance of PB, a sender broadcasts a value v and an associated *proof* to all n parties, which validate the first v they receive using the proof and an *external validity* function EV, sign v with their own keys and echo the signed v to the sender, which collates signatures from $n - f$ parties into a *delivery certificate* for v. As Abraham *et al.* observe [4], an iterated version of PB forms the main loop of Tendermint [15], Casper [17], and HotStuff [65]. The

first iteration (if it terminates) proves that $n - f$ parties know and have validated v; the second proves that $n - f$ parties know that $n - f$ parties know and validated v, and so on.

Following Lamport's classification [35], a protocol's correctness specification can be divided into *safety* properties, which assert that "bad states" cannot be reached, and *liveness* properties, which assert that "good states" are eventually reached. Typically, the satisfaction of these properties is conditioned on the number of faults the system experiences. In a Byzantine consensus protocol, for instance, data integrity can be guaranteed in an asynchronous setting with f faulty nodes only if at least $2f+1$ nodes are non-faulty (*i.e.*, $n > 3f$); eventual termination can be guaranteed as well when assuming certain fairness condition. Provable Broadcast satisfies three safety properties and one liveness property:

(1) *Weak Availability* (S): if a delivery certificate exists for v, then at least $n - 2f$ honest parties hold and echoed the value v;
(2) *Uniqueness* (S): at most one value obtains a delivery certificate;
(3) *External Validity* (S): if there is a delivery certificate for a value v, then v is externally valid;
(4) *Termination* (L): an honest sender that has an externally valid input v will eventually obtain a delivery certificate for v.

Weak availability follows from the construction of the protocol. Uniqueness follows from quorum intersection, by contradiction: having two delivery certificates for different values $v \neq v'$ implies there are two sets of $n - f$ parties who have signed v and v', respectively. But if $n > 3f$, these two sets intersect in at least $f + 1$ parties, and thus one honest party signed different values, which is impossible for honest nodes. External validity follows from weak availability and the fact that all honest parties validate the echoed value. Finally, termination follows under the fairness condition that packets between honest nodes are eventually delivered.

Iterated versions of PB, in which v and the delivery certificate from iteration i become the value and the proof respectively in iteration $i + 1$, provide *increasingly stronger* guarantees. Concretely, by combining the weak availability of iteration $i+1$ with the uniqueness of iteration i, we prove properties of the form "a quorum knows that a quorum knows v", with increasing depth. This "locks" values such that nodes cannot later renege them. Such guarantees are the basic logical building block of BFT consensus protocols.

Problem statement. Assume one would like to establish a stronger property for the PB protocol iterated *two* times, a composition known as *Locked Broadcast* (LB). For example, consider *Unique Lock Availability*: if a delivery certificate exists for v then it implies there exists a *lock certificate* for v, which no other value v' can have, and there are at least $n-2f$ honest parties that hold this lock certificate; the lock certificate implies that at least $n - 2f$ honest parties hold the value v. This property follows from uniqueness (2) of PB run in LB's first stage and weak availability (1) of LB's second stage.

While intuitive and simple, a compositional proof of this property *out of the already established facts* (1) and (2) cannot be obtained in most of the modern frameworks for verifying distributed protocols. While some existing approaches allow for deriving properties of composite distributed protocols from their sub-parts, they either lack support for liveness proofs [54, 62] or do not allow reasoning about Byzantine behaviours [29, 30], or both [57, 63]. Recent works that formally address both liveness properties and Byzantine fault tolerance either focus on a concrete monolithic blockchain system in a synchronous network [61] or lack the ability to compose proofs across protocols [10, 42, 53]. The only approach to date we are aware of that is capable of the desired proof decomposition [32], requires encoding a system as a *threshold automaton* suitable for model checking [33], thus giving up on the expressivity of the specification formalism and on the foundational verification guarantees.

The goal of this work is to provide a framework for machine-assisted verification of BFT protocols, which supports proofs of their safety and liveness, that are both compositional (*i.e.*, are constructed out of properties of sub-protocols) and foundational (*i.e.*, do not require a bespoke trusted encoding of the system).

This work. In our quest, we are taking forward the ideas from earlier efforts on encoding and verifying distributed systems in interactive proof assistants based on higher-order logic [50, 54, 57, 63], enhancing them with the following new aspects:

- *Knowledge lemmas.* For each verified basic BFT protocol, we identify and prove a family of lemmas, which provide a succinct summary of the knowledge derivable from any execution history of a protocol, facilitating the proofs of safety and liveness. While the lemmas have to be stated in terms of the data types specific to a protocol's definition, their shapes are generic, making their statements easy to formulate for any BFT construction.
- *A functor for protocol composition.* We provide a higher-order functor to build composite protocols from basic sub-protocols. This structure allows for the safety and liveness properties of a composite protocol to be derived from those of its sub-protocols.

The result of these enhancements is Bythos, a new framework embedded in the Coq proof assistant [60] for the compositional verification of both safety and liveness properties of BFT protocols.

Contributions. Our work makes the following contributions:

- We present a methodology for compositional verification of BFT protocols based on modular proofs of inductive invariants and temporal reasoning, that applies to a wide range of basic protocol constructions and their combinations.
- We introduce the idea of *knowledge lemmas*: building blocks of inductive invariants that concisely encapsulate the causality of the system execution, facilitating its safety and liveness proofs.
- We propose the systematic decomposition of composite BFT protocols and verification of their properties, especially liveness ones, by modularising the protocol implementations and specifications using the mechanism of Coq modules and functors.
- We implement our methodology as Bythos, a domain-specific language and library of lemmas in the Coq proof assistant [60]. To the best of our knowledge, this is the first verification framework that (1) is foundational, (2) allows reasoning about Byzantine faults, (3) supports both safety and liveness proofs, and (4) allows for effective proof reuse for composite BFT protocols.
- We showcase Bythos and its compositional nature by verifying the Provable Broadcast (PB) protocol [55] and two of its iterated versions [2], Bracha's Reliable Broadcast (RB) protocol [14], and the recent Accountable Confirmer (AC) protocol [22], as well as the composition of RB and AC. Ours are the first machine-checked formalisations of any of these protocols.

```
1  Record State := Node {
2    id : Address;
3    (* sender state *)
4    sent : Round -> bool;
5    counter : Round -> list (Address * PartialSignature);
6    output : Round -> option CombinedSignature;
7    (* receiver state *)
8    echoed : Address * Round -> option (Value * Proof) }.
9  Inductive InternalEvent :=
10   | SendAction (r : Round).
11 Inductive Message :=
12   | Init (r : Round) (v : Value) (pf : Proof)
13   | Echo (r : Round) (ps : PartialSignature).
```

(a) Local state defined as a record, with internal event and message defined as algebraic datatypes, where `Round`, `Value` and `Proof` are all declared as `Parameter`s.

```
1  Definition procMsg st src msg : State * list Packet :=
2    let: Node q _ cnt output _ := st in
3    match msg with
4    | Echo r ps => if (output r == None) &&
5          (partial_verify (r, proposal q r) ps src) &&
6          (src ∉ (map fst (cnt r))) then
7        let: cnt' := {cnt[r] ↦ (src, ps) :: cnt r} in
8        let: st' := if length (cnt' r) == n - f then
9          let: cs := partialsig_combine (map snd (cnt' r)) in
10         st <| output := {output[r] ↦ Some cs}, cnt := cnt' |>
11       else st <| cnt := cnt' |> in (st', [])
12     else (st, [])
13   | Init r v pf => (* processing the Init message *)
```

(b) Condensed version of the message handler, `procMsg`. We use `{mp[a] ↦ b}` to denote a map update and `r <| f := ... |>` to denote a record field update.

Fig. 1: Highlights from the Bythos encoding of Provable Broadcast.

Paper outline. Sec. 2 gives a tour of Bythos from the user's perspective, demonstrating its features by following the formalisation of Provable Broadcast, its correctness proofs, and the verification of two of its iterated versions, outlining the conceptual idea of knowledge-driven proofs of safety and liveness in Sec. 2.2 and Sec. 2.3, respectively. Subsequently, Sec. 3 presents the implementation of Bythos on top of Coq, focusing on the components the user must instantiate to define a working system (Sec. 3.1), the support for liveness specifications and proofs (Sec. 3.2) and the functor for composing protocols (Sec. 3.3). Sec. 4 describes more case studies: verification of Reliable Broadcast (RB, Sec. 4.1), Accountable Confirmer (AC, Sec. 4.2), and Accountable Reliable Broadcast—the composition of RB and AC (ARB, Sec. 4.3). Sec. 5 discusses the proof efforts of this work and the limitations of Bythos. Sec. 6 compares Bythos with related frameworks for mechanised verification of safety and liveness of distributed protocols.

2 Overview

In this section, we take the Provable Broadcast protocol as a running example to showcase the process of implementing BFT protocols and verifying their safety and liveness in Bythos, and highlight how to compose protocols to obtain strengthened guarantees.

2.1 Provable Broadcast, Formally

In Bythos, a protocol is defined in terms of two *handlers*, implemented as pure functional programs:

- `procInt`, which handles *internal events* within a node, *e.g.* the decision of the sender in PB to broadcast a value;
- `procMsg`, which defines the effect of receiving a message.

Each of them takes as input the local state of a node and, respectively, the internal event or message to be handled, and returns the updated state and a list of packets (*i.e.*, messages decorated with sender's and receiver's identities) to be sent in response.

We start implementing the Provable Broadcast protocol by defining a node's local state, the kinds of internal events that can occur, and the type of messages, as shown in Fig. 1a. The only internal event in PB is the initiation of a broadcast (line 10), where a sender sends to all nodes `Init` messages, each of which contains a value and a proof (line 12). The receivers echo by sending `Echo` messages that contain signatures (line 13). Unlike the simplified description [1–3], in scenarios where PB is used as a sub-protocol (*e.g.*, consensus protocols like [4]), each node acting as the sender can initiate and concurrently execute multiple rounds of PB, and all messages thereby contain `Round` tags for disambiguation (lines 12–13). When echoing, nodes sign a round-value pair rather than just the value. Because a node can be part of multiple concurrent rounds, acting in different capacities in each, nodes maintain state both for acting as a sender (lines 4–6) and acting as a receiver (line 8).

Every node has a fixed unique address (`id` on line 2), *i.e.*, a cryptographic public key that for simplicity also doubles as a network identifier to which packets can be sent. For each round, the sender keeps track of whether it broadcast a value in that round (`sent` on line 4), of which signed echoes it received (`counter` on line 5), and whether it output a delivery certificate (`output` on line 6). Receivers keep track of what they have echoed to whom and at which round (`echoed` on line 8). We model the values to broadcast as taken from `proposal` (used on line 5 in Fig. 1b), an oracle we take as a parameter of the protocol, defining what value node q proposes at round r.

2.1.1 Specification of Cryptographic Primitives. In Bythos, we work in the Dolev-Yao model [24]. Specifically, we assume ideal cryptographic primitives from the threshold signature scheme [41], with the following standard axioms, often assumed in hand-written distributed protocol proofs [4]:

- A *partial signature*[1] ps can be verified *wrt.* the pair (r, v) and the public key q if and only if ps is signed for (r, v) using the corresponding private key of q.
- A *combined signature* cs can be verified *wrt.* the pair (r, v) if and only if cs is obtained from collating $n-f$ partial signatures, where for each partial signature ps, there must exist a public key q such that ps can be verified with respect to (r, v) and q, and these q are different for each partial signature.
- The signing function is injective. That is, any given partial signature ps can only be verified with respect to a single input (r, v).

The `PartialSignature` and `CombinedSignature` types (*cf.* lines 5, 6, and 13 of Fig. 1a), which abstract partial signatures and combined signatures respectively, are predefined as `Parameter`s in Bythos. Their associated methods including `partial_sign`, `partial_verify`

[1] Also called *signature share* [4]. Here, we follow the terminology by Civit *et al.* [22].

(line 5 in Fig. 1b), `partialsig_combine` (line 9), and `combined_verify` are also declared as **`Parameter`**s satisfying the axioms listed above.

2.1.2 Protocol Definition. With these preliminaries, we can define the two handlers. In the interest of space, in Fig. 1b we show only a condensed version of the `Echo` message handler at the sender, which receives (line 4), verifies (line 5), and records (line 7) signatures from the protocol parties, and when the threshold of $n - f \geq 2f + 1$ valid partial signatures received from distinct parties is reached (line 8), builds the combined delivery certificate (line 9). In particular, on line 5, the sender (with identifier q) verifies that the value signed is indeed the value that it originally proposed. The `Init` message handler at the receiver is defined similarly: if the attached value and proof are validated by the external validity function EV, then the receiver records them into the `echoed` field, uses `partial_sign` to sign the round-value pair and attaches the signature to the `Echo` that is then sent back. The `procInt` handler for `SendAction` uses `proposal` to choose a value and broadcasts it by returning a list of packets rather than `[]`, as `procMsg` does on line 12.

2.2 Proving Safety Properties

With a full implementation of the protocol, we proceed to verify that our implementation satisfies the four properties listed in Sec. 1. But first we must understand how to formally express these properties and, more fundamentally, *what it means* for a protocol to satisfy a property. Both aspects are tied to the *system model* of Bythos.

2.2.1 System Model. In Bythos, the entire distributed system, including all nodes and the network, is modelled as a state-transition system. The `SystemState` consists of a mapping from each node's `Address` to its local (protocol-specific) `State`, as well as a collection of *all* the packets that have been ever sent, called the *packet soup*. In the packet soup, packets that have been *received* (i.e., delivered by the network) are distinguished from those that have not, which are referred to as being *fresh*. The system has four different kinds of transitions, or steps, which define how a system state can change:

- *Stuttering*, which leaves the system state unchanged. The existence of such a step is essential for compositionally reasoning about composite protocols (*cf.* Sec. 4.3).
- *Delivery*, where a non-faulty node receives some packet in the packet soup. As mentioned in Sec. 2.1, `procMsg` is for handling the packet and defines what the node's local state should be updated to, as well as the packets that should be added to the packet soup.
- *Internal*, where a non-faulty node spontaneously triggers an internal event, without input from the network. The `procInt` handler specifies how the internal event should update the node's local state and which, if any, freshly produced packets should be added to the packet soup.
- *Byzantine*, where a Byzantine node adds a single fresh packet to the packet soup. The packet's source must be one of the Byzantine nodes (*i.e.*, packets cannot be forged), and the message contained within the packet must satisfy a protocol-specific predicate `byzConstraints`, which specifies the messages that the adversary can produce in a given system state (more details in Sec. 3.1.5).

Packets are never removed from the packet soup. Moreover, the fact that we distinguish between fresh and received packets plays an important role in our proofs. The intuition is that packets serve as *mediators* that induce causal relationships: the fact that a packet was sent (*i.e.*, it is in the packet soup) implies something about the local state of the sender s, and similarly, a packet's receipt has implications for the state of the receiver r. Put together, the existence of a received packet lets us relate the states of s and r, which would be difficult if we were unable to determine whether the packet was received; an instance of such correlation is property (5a) in Sec. 2.2.5. Such discrimination over packets is widely used in other verification frameworks [30, 46, 57].

```
1  Definition producible p r v (cs : CombinedSignature)
2    (w : SystemState) : Prop :=
3      forall (pss : list PartialSignature),
4      cs = partialsig_combine pss ->
5      forall (q : Address) (ps : PartialSignature),
6        ps ∈ pss ->
7        (* q is honest *) ->
8        partial_verify (r, v) ps q = true ->
9        exists pkt : Packet,
10         (* pkt is in the packet soup, is sent by node q,
11            and carries the message (Echo r ps) *).
```

Fig. 2: Predicate constraining any producible delivery certificate `cs` by node `p` for round `r` and value `v`, in system state `w`.

2.2.2 Modelling Byzantine Faults. We model the Byzantine adversary as controlling the asynchronous network, having the ability to observe all packets and indefinitely delay packets. However, the adversary cannot tamper with the contents of packets between honest nodes and cannot forge messages to appear as coming from honest nodes. Moreover, the adversary is constrained in terms of what values it can *produce*, regardless of whether these values become externally visible via messages or not.

To state the assumption that Byzantine nodes do not possess unbounded computing power and thus cannot forge the signatures of non-faulty nodes, we follow the Dolev-Yao approach [24] and state that the only way Byzantine nodes can acquire signatures signed with the private keys of non-faulty nodes is by intercepting packets transmitted over the network. In the context of Provable Broadcast, the adversary is constrained in terms of which delivery certificates (*i.e.*, combined signatures) it can produce. We formalise this as shown in Fig. 2. The `producible` predicate says that if `cs` is a producible certificate in the system state `w` obtained by combining a list of partial signatures `pss` (which is the only way to form a combined signature that can be verified as per the axiom in Sec. 2.1.1), then all valid signatures in `pss` from honest nodes were seen in the history of `w` (*i.e.*, in its packet soup) as part of `Echo` messages. Intuitively, this predicate captures all the possible ways a Byzantine adversary could combine partial signatures into delivery certificates—the existence of any other delivery certificate implies a break of the cryptography. Taking this constraint as premise in the proof of uniqueness (2) suffices to establish the safety and liveness of the system, without additional constraints on which messages can be sent. Consequently, the `byzConstraints` predicate is simply `True` for Provable Broadcast.

2.2.3 Safety Properties. Having defined the system model, we can now state what a property is and what it means for a system to

satisfy a property. A *safety* property is a predicate on `SystemStates`. A system satisfies a safety property S if every system state w reachable via valid system steps from an initial system state satisfies S. The standard approach to prove safety properties is to establish an *inductive invariant*, a property I which is closed under (*i.e.* preserved by) system transitions. Once we find an inductive invariant I such that $\forall w, I(w) \implies S(w)$ and show that I holds on the initial system state, we can establish that the system satisfies S.

2.2.4 Knowledge Lemmas as Incremental Invariants. However, it is usually difficult to come up with an inductive invariant all at once. As a rough intuition, an inductive invariant reflects some knowledge about the protocol execution "contained" within a system state and preserved under transitions. If the invariant is to imply any interesting safety property, it must capture a lot of information about the protocol's execution. But protocols themselves are not monolithic, and the properties they guarantee do not "fall from the sky". Rather, protocols are designed in a gradual fashion, such that high-level guarantees are built progressively on top of low-level guarantees. This is a form of logical composition within a single protocol, and arises naturally in well-designed systems. And we ought to verify protocols in the same way they are designed. This is the intuition behind our idea of *knowledge lemmas*: we want to *incrementally* capture the knowledge within a distributed execution following the way this knowledge is generated in the first place.

2.2.5 Knowledge-Driven Proof of Safety. This process of incremental construction of an inductive invariant is best understood with a worked example at hand. We build our inductive invariant as the conjunction of knowledge lemmas from each of the following categories, with representative knowledge lemmas shown for each category (please refer to the definitions in Fig. 1a):

(1) **Data representation:** every list of collected signatures stored in `counter` does not contain duplicates, and every included signature verifies; every value-proof pair collected in `echoed` passes the external validity function;
(2) **Data persistence:** the `echoed` field, once set for an address and round, is never overwritten; similarly for the `output` field; the `counter` field only grows; the `id` field remains constant;
(3) **Knowledge propagation within a node:** if `(output r)` is set for some round `r`, then `(counter r)` has length exactly $n - f$ and `(output r)` is obtained by combining the partial signatures in `(counter r)`; if it is not set, then the length is less than $n - f$;
(4) **Knowledge propagation through packets**, where there are two *independent* attributes:
   (a) **Direction:** if a node q records that it initiated round r, then there is an `Init` message for r sent by q to every node in the packet soup, and symmetrically, if there is an `Init` message from node q, then q's local state has `(sent r)` being `true` and that message carries q's proposal for round r; these two represent knowledge propagation in the **local-to-packet** and **packet-to-local** directions, respectively;
   (b) **Receipt-sensitivity:** if a node q records in `echoed` that it echoed to node p for round r, then there is a received `Init` message for r sent by p in the packet soup; conversely, if there is a received `Init` message containing an externally valid value for r sent by p to an honest node q, then q *must* have its `echoed` field set for p and r (q may not have echoed to the same `Init` message, though, since p can be Byzantine and have sent multiple such `Init` messages); these two represent **receipt-sensitive** knowledge propagation, requiring that the described message be received, while those in (4a) are **receipt-insensitive**, which hold regardless of whether the message is received;
(5) **Implications of knowledge:**
   (a) *Echoed implies proposed:* if an honest node a echoed to another honest node b, then it echoed what b proposed;
   (b) *Counted implies echoed:* if an honest node a has a partial signature from another honest node b in its `(counter r)`, then b must have echoed the same value that a initially proposed for round r, and that value is externally valid;
   (c) *Honest output:* if an honest node output a delivery certificate for v, then the certificate can be verified and at least $n - 2f$ honest nodes echoed v; moreover, v is externally valid;
   (d) *Uniqueness:* in any instance of Provable Broadcast (determined by the sender and the round), every producible certificate is for the same value.

The knowledge lemmas from categories (1)–(4) capture the low-level properties of the protocol, which directly follow from the protocol design and the system model. Specifically, the first three categories of knowledge lemmas focus solely on the local states of honest nodes, derived from the "delta" between the local state updated by `procInt` or `procMsg` and the original one. On the other hand, knowledge lemmas from category (4) describe the mutual effect between the local states of honest nodes and packets sent from or to them.

By composing existing knowledge, we can then incrementally devise higher-level guarantees, as those listed in category (5). For example, property (5a) can be derived by composing the first property in (4b) and the second one in (4a). Moreover, property (5a) itself can be used in proving property (5b), which, in turn, aids in proving other properties. In this way, we establish the safety of Provable Broadcast: property (5c) implies weak availability and external validity, and property (5d) directly is uniqueness.

2.3 Reasoning about Liveness

2.3.1 Liveness Properties. Whereas safety properties are predicates over states, liveness properties are predicates over *infinite execution traces* of system states, usually expressed in *temporal logic* [52]. To encode and prove liveness properties in Coq using temporal logic, we employ the CoqTLA library [20]. While it is straightforward to define how a system satisfies a safety property (*cf.* Sec. 2.2.3), it is much trickier to do so for a liveness property. If we require that any possible execution satisfies the property, then effectively no non-trivial properties could be satisfied by a system: an execution consisting of only stuttering steps does not do anything interesting. Rather, the properties we want to prove hold only over "reasonable" executions. In the literature, this concept of reasonableness is described in terms of *fairness conditions*, restrictions over executions. A system satisfies a liveness property if all *fair* executions of the system satisfy the property.

2.3.2 Fair Delivery. Our fairness assumption cannot be related to clocks or Byzantine nodes due to the presence of asynchrony and

the Byzantine adversary. Therefore, the fairness assumption we adopt as default throughout Bythos is *fair delivery*, which states that all packets between honest nodes are eventually delivered.

2.3.3 Knowledge-Driven Proofs of Liveness. We prove liveness by first identifying, for the sake of modularising the proof, a decomposition of the protocol into *phases*. These phases arise naturally in asynchronous protocols, resembling "rounds" in synchronous protocols. Then, proving liveness amounts to showing that under the fair delivery assumption, these phases are guaranteed to happen consecutively, with the start of the first phase "leading to" the end of the last phase (a formal notion of "leading-to" is given in Sec. 3.2).

Consider an instance of Provable Broadcast initiated by an honest node q for round r, where q is going to propose an externally valid value v. The natural decomposition is into two phases: an "initial" phase in which `Init` messages circulate, and an "echo" phase in which `Echo` messages circulate. This intuitive description is not quite fit for formalisation, however. We need to be more precise. Concretely, we define the start and end of each phase to be predicates over system states: (1) the "initial" phase starts when q locally sets `(sent r)` to `true`; (2) the "initial" phase ends, as well as the "echo" phase starts, when *all* honest nodes `echoed` for q and r; (3) the "echo" phase ends when q produces a delivery certificate for v. To show that both phases will eventually conclude, we conduct the following reasoning, utilising the fairness assumption and a series of knowledge lemmas established in Sec. 2.2.5:

(i) By the first knowledge lemma from (4a), when the "initial" phase starts, q has sent to all nodes `Init` messages that carry v and which can be found in the packet soup;
(ii) By the fair delivery assumption, all honest nodes will eventually receive these `Init` messages;
(iii) At that time, by the second knowledge lemma from (4b), all honest nodes will have their `echoed` fields set for q and r;
(iv) Moreover, by property (5a), all honest nodes will have `echoed` to the `Init` messages carrying v, indicating the end of the "initial" phase and the start of the "echo" phase;
(v) By a local-to-packet receipt-insensitive knowledge lemma and the fair delivery assumption, q will eventually receive `Echo` messages from all honest nodes in response to its previously sent `Init` messages;
(vi) At that time, by a packet-to-local receipt-sensitive knowledge lemma, q will have collected all partial signatures on those `Echo` messages to its `(counter r)`;
(vii) Finally, by the knowledge lemmas from (3) and property (5c), since the number of honest nodes is larger than $n - f$, q will have produced a delivery certificate for v by that time.

The *termination* property is then proved by following the reasoning steps above. (Its formal statement is shown in Fig. 4 and will be explained in Sec. 3.2.)

We note that the key element in the liveness proof is the application of packet-to-local receipt-sensitive knowledge lemmas in steps (iii) and (vi). Intuitively, those lemmas state that honest nodes will not "reject" well-formed messages delivered to them; without such a guarantee, we cannot apply further knowledge propagation.

2.4 Verifying Protocol Composition

We conclude our overview by demonstrating how to verify the safety of iterated versions of the Provable Broadcast protocol, in a very simple way. For composable liveness proofs, see Sec. 4.3.1.

If we run two iterations of Provable Broadcast one after the other, taking the output of the first as the input of the second, we obtain a stronger safety property called *Unique Lock Availability*, *cf.* Sec. 1. Very interestingly, we can prove the safety of the iterated protocol in only 5 lines of Coq, by creating two instances of the Coq `Module` that defines Provable Broadcast, and simply adding a hypothesis that states that both instances propose the same value and the proof used in the second (*cf.* line 12 in Fig. 1a) is the delivery certificate output at the end of the first instance. The proof then consists simply of directly applying the safety properties of the individual instances. We can then add yet another iteration of PB, and obtain an even stronger guarantee with the same ease.

3 Bythos Under the Hood

We implemented Bythos as a shallow embedding into the language and logic of the Coq proof assistant [60]. The framework comes with a set of reusable definitions for Byzantine network semantics, generic shape of system state, helper lemmas for discharging common safety and liveness proof obligations, as well as a functor (*i.e.*, a higher-order Coq module) for constructing composite protocols, which we will outline in Sec. 3.3. To keep things simple for the purpose of protocol modelling, Bythos does not support advanced language features, such as concurrency, exceptions, or advanced control operators, unlike some existing program logics for safety verification of distributed systems [34, 57, 58]. In particular nodes in protocols defined in Bythos do not feature any additional mutable state besides what is explicitly declared in their local state data type (*e.g.*, the `sent` field in Fig. 1a for the case of Provable Broadcast).

In this section, we will elaborate on the components of Bythos that a user needs to instantiate to define a complete distributed system acting in a Byzantine environment in a way that is amenable for safety proofs (Sec. 3.1), enables stating and proving liveness properties (Sec. 3.2), and allows for composing individually defined protocols into composite systems (Sec. 3.3). We will also briefly illustrate how to make protocol models in Bythos executable (Sec. 3.4).

3.1 Instantiating a Byzantine System in Bythos

3.1.1 Basic Data Types. As demonstrated by the example in Fig. 1a, every user-defined protocol in Bythos is expected to provide two concrete data types: `Address` and `Message`, both used to instantiate the parametric definition of the network semantics, which we will show in Sec. 3.1.6. To facilitate the proofs, Bythos requires `Address` to be inhabited and finite, which matches the real-world scenario where the set of nodes in a protocol is not empty but is nonetheless finite. The derived datatype of packets, `Packet` is defined as a Coq `Record` parameterised by `Address` and `Message`:

```
Record Packet := mkP { src : Address; dst : Address;
                       msg : Message; received : bool }.
```

This definition decorates its wrapped message `msg` with sender `src` and receiver `dst` identities, as well as a Boolean tag `received` indicating whether this packet has been delivered.

Bythos also provides abstractions of ideal cryptographic primitives, with their interfaces specified and their guarantees axiomatised (*e.g.*, those listed in Sec. 2.1.1).

3.1.2 Byzantine Resilience Threshold. As the next parameter of the system, the user needs to specify the maximum number of Byzantine nodes `f` that the protocol can tolerate as a function of its number of nodes n. Bythos requires that `f` be smaller than n, imposing a (typically trivial) proof obligation on the user. For instance, for many protocols, `f` is $\lfloor (n-1)/3 \rfloor$. To account for this very common scenario, Bythos provides a pre-defined system template, where `f` is fixed to be $\lfloor (n-1)/3 \rfloor$ and some useful lemmas (*e.g.*, properties about quorums of size $n - f$). This template is useful for verifying protocols with the *optimal resilience* [14].

3.1.3 Protocol. With the basic data types and the Byzantine resilience threshold given, the next step is to define the actual protocol. This is achieved by instantiating the local state datatype `State`, the kinds of internal events `InternalEvent` and the two handlers `procInt` and `procMsg`, as demonstrated in the example from Fig. 1 (although most protocols have more complex definitions).

3.1.4 System State. Recall from Sec. 2.2.1 that the system state includes a mapping of each node's identity to its local state, along with a packet soup. Bythos represents `SystemState` naturally as a record type, with the user having to provide the definition `initState` of individual nodes' initial local states to instantiate the initial overall system state `initSystemState`, as shown below:

```
Record SystemState := mkW { localState : Address -> State;
                            packetSoup : list Packet }.
Definition initSystemState : SystemState := mkW initState [].
```

For simplicity, in Bythos the packet soup is defined as a list of packets, which is initially empty. Whether a packet in the packet soup is received can be determined via its `received` field.

3.1.5 Byzantine Nodes and Their Behaviour. Defining the semantics of the entire system requires specifying which nodes are acting in a Byzantine fashion and precisely modelling their behaviour. We specify which nodes are malicious through the `isByz` system parameter of type `Address -> bool`, which returns `true` if the argument is a Byzantine node. Unlike previous system components (*e.g.*, the `Message` data type or the `initState` value), `isByz` does not have to be instantiated, and is instead treated as a universally quantified variable in all proofs about the system, since the system's properties should hold under *any* setting of Byzantine nodes. Naturally, unlike other parameters, `isByz` must not be instantiated when extracting executable system implementations, as implementations have no control over the number and identities of Byzantine parties.

A key aspect of describing the behaviour of Byzantine nodes is specifying what messages they can send, which is determined by the adversary's capability. To model this capability, Bythos requires the user to instantiate

```
Parameter byzConstraints : Message -> SystemState -> Prop.
```

which constrains the messages that can be sent by a Byzantine node during executions of the system. Although this predicate has a simple type, we can model adversaries with varying degrees of power by instantiating it differently. For example, by instantiating the predicate to be `(fun _ _ => True)` in Coq, we model the strongest possible adversary, one that is able to send *any* messages during the protocol's execution. This is exactly the assumption made in the proofs of the Reliable Broadcast protocol discussed further in Sec. 4.1. In particular such an adversary can send a message that breaks the security guarantees of cryptographic primitives (*e.g.*, a message with a signature that requires the private key of a non-faulty node), and thus the adversary can be viewed as possessing unbounded computational power and even the knowledge of the local states and private keys of non-faulty nodes. More commonly, we instantiate `byzConstraints` to model a more realistic adversary following the Dolev-Yao model [24], allowing a Byzantine node to produce a message based solely on the knowledge obtained from intercepting packets in the packet soup.

In some proofs, we also need to constrain Byzantine nodes in terms of what values they can produce (potentially not visible via messages), as in Sec. 2.2.2. Such constraints are expressed in the same way as `byzConstraints` and typically taken as premises in the statements of key protocol properties.

Stuttering

$$\frac{}{(\Sigma, PS) \xrightarrow{\mathrm{S}} (\Sigma, PS)}$$

Internal

$$\frac{\texttt{isByz}\ p = \texttt{false} \qquad \texttt{procInt}\ \Sigma(p)\ \ ev = (st, pkts)}{(\Sigma, PS) \xrightarrow{\mathrm{I}(p,ev)} (\Sigma[p \mapsto st], \texttt{sendout}(pkts, PS))}$$

Delivery

$$\frac{p = pkt.\texttt{dst} \qquad \texttt{isByz}\ p = \texttt{false} \qquad pkt \in PS \qquad \texttt{procMsg}\ \Sigma(p)\ \ pkt.\texttt{src}\ \ pkt.\texttt{msg} = (st, pkts)}{(\Sigma, PS) \xrightarrow{\mathrm{D}(pkt)} (\Sigma[p \mapsto st], \texttt{sendout}(pkts, \texttt{consume}(pkt, PS)))}$$

Byzantine

$$\frac{\texttt{isByz}\ p = \texttt{true} \qquad pkt.\texttt{received} = \texttt{false} \qquad \texttt{byzConstraints}\ \ pkt.\texttt{msg}\ \ (\Sigma, PS)}{(\Sigma, PS) \xrightarrow{\mathrm{B}(pkt)} (\Sigma, \texttt{sendout}([pkt], PS))}$$

where

$$\Sigma[p \mapsto st](q) \stackrel{\text{def}}{=} \begin{cases} st, & \text{if } q = p \\ \Sigma(q), & \text{otherwise} \end{cases}$$

$$\forall pkt, pkt \in \texttt{sendout}(pkts, PS) \iff pkt \in pkts \vee pkt \in PS$$

$$\forall pkt, pkt \in \texttt{consume}(pkt', PS) \iff pkt = \texttt{markRcv}\ \ pkt' \vee (pkt \in PS \wedge pkt \neq pkt').$$

Fig. 3: System semantics. A system state is a pair (Σ, PS) of `localState` and `packetSoup`, respectively. A transition step from (Σ, PS) to (Σ', PS') is represented as $(\Sigma, PS) \longrightarrow (\Sigma', PS')$. Each step is associated with a tag (*e.g.*, $\mathrm{D}(pkt)$, $\mathrm{B}(pkt)$), that carries additional information about the step (*e.g.*, which packet is delivered) and indicates which type of transition is taken. The function `markRcv pkt` returns `pkt <| received := true |>`.

3.1.6 System Semantics. With the definitions and parameters from Sec. 3.1.1–3.1.5, we can instantiate the system semantics of Bythos for a particular protocol. The generic definition of the semantics parametrised over those definitions is given in Fig. 3; in our Coq implementation, we model it as a functor taking a set of modules defining the above described components as input. The functor defines an inductive relation with four constructors, replicating the logic of the semantic rules from Fig. 3, which we detail below.

The rules for Stuttering and Internal steps are straightforward. In the case of the Delivery step, the packet soup after transitioning must contain the packets produced by the protocol's `procMsg`, and also mark the packet delivered at this step as received. To this end, we use the combination of two functions, `sendout` and `consume`, to achieve this: informally, `sendout`($pkts, PS$) adds the packets $pkts$ to the packet soup PS, and `consume`(pkt', PS) marks the packet pkt' inside PS to be received. The Byzantine step adds an arbitrary packet to the packet soup, ensuring that its contents do not violate the restrictions imposed by `byzConstraints`. Since Byzantine nodes may not conform to the protocol, we do not specify how their local states change. Additionally, we do not consider whether packets will be received by Byzantine nodes, as reflected in the `isByz` p = `false` premise of the Delivery step.

The generic definition of the system semantics in Fig. 3 accurately captures our basic assumptions. For instance, we assume the network model to be asynchronous, since there is no internal constraint on *when* a Delivery step will be taken, which captures the scenario of packets being arbitrarily rearranged or delayed by the adversary. The semantics also allows the network to deliver the same packet multiple times by not requiring pkt.`received` to be `false` in the $pkt \in PS$ premise of the Delivery step rule. This models the assumption that packets may duplicate. Our semantics does not encode tampering or loss of packets explicitly, as those can be modelled by a suitable choice of an adversary in combination with the non-deterministic nature of our semantics.

Lastly, we note that despite its conceptual simplicity the semantics in Fig. 3 still allows us to prove several generic facts that we can use in reasoning about execution properties of arbitrary protocols. One example of such property is *step locality*, which follows from the fact that the system semantics treats the execution of handlers or the step of a Byzantine node as being *atomic*: for the Internal or the Delivery step, only one node will be executing `procInt` or `procMsg`, and its execution is completed in one transition step; for the Byzantine step, only one Byzantine node is allowed to send a packet in a transition step. Therefore, *at most one* node will have its local state changed after transition. Another useful generic property is *soup growth monotonicity*, which has been observed and used in prior works on formal verification [5]: if a received packet is in the soup, then that packet will remain in it after any transition, which implies that the packet soup only grows.

3.2 Specifying and Proving Liveness

One of the main features of Bythos is the ability to express liveness properties using the connectives from Linear Temporal Logic (LTL) [52] and prove them using the rules of Lamport's Temporal Logic of Actions (TLA) [36]. To achieve that, we have incorporated the CoqTLA library [20] into Bythos by developing a Coq functor

```
Definition pb_termination : Prop := forall q r v,
  isByz q = false -> v = proposal q r ->
  (* v is externally valid *) ->
  ⌜ init ⌝ ∧ □ ⟨ next ⟩ ∧ WFDelivery ⊢
  ⌜ initiated q r ⌝ ⇝ ⌜ has_certificate_for q r v ⌝.
```

Fig. 4: PB termination statement in Bythos via CoqTLA.

that takes a module implementing a system instance (assembled as outlined in Sec. 3.1) as an argument and adapts its definitions to the TLA notation and semantics.

With the help of the LTL notations provided by CoqTLA, we can express the same properties in Bythos in a manner almost identical to the expressions of liveness properties in TLA, allowing us to state liveness properties in an intuitive way. As an example, consider the *termination* property of the Provable Broadcast protocol (Sec. 1) stating that if an honest node q broadcasts some externally valid value v at round r, then *eventually* it will output a delivery certificate for v. Its statement in Bythos is shown in Fig. 4, with some definitional details omitted for brevity.

The embedding of Bythos system semantics into CoqTLA is made possible thanks to the *polymorphic* nature of the definitions in CoqTLA, which allows the logical connectives to work immediately for predicates on traces of system states produced by Bythos. To understand the notations from Fig. 4, recall that in TLA, a trace σ of system states is defined as a function from natural numbers (representing timestamps) to system states. For a predicate P over system states, $\sigma \vDash \ulcorner P \urcorner$ if $P(\sigma(0))$ holds; for a binary predicate A over system states (so-called *action* in TLA), $\sigma \vDash \langle A \rangle$ if $A(\sigma(0), \sigma(1))$ holds. Furthermore, by the semantics of the *always* temporal operator, $\sigma \vDash \Box\langle A \rangle$ if for any $i \in \mathbb{N}$, $A(\sigma(i), \sigma(i+1))$ holds. Therefore, we can encode the fact that σ is indeed a trace produced via consecutive system transitions by requiring $\sigma \vDash \ulcorner \texttt{init} \urcorner \wedge \Box\langle \texttt{next} \rangle$, where $\texttt{init}(w) \stackrel{\text{def}}{=} (w = \texttt{initSystemState})$ and the transition relation `next` is defined as $\texttt{next}(w, w') \stackrel{\text{def}}{=} w \longrightarrow w'$ (*cf.* Fig. 3).

In addition to basic definitions like `init` and `next`, the Bythos functor for enabling temporal specifications also provides a tailored notion of fairness, denoted as `WFDelivery`. As we will show below, under certain conditions, `WFDelivery` is logically equivalent to the common eventual delivery assumption (Sec. 2.3.2) used in our liveness proofs. Formally, `WFDelivery` is defined as

$$\texttt{WFDelivery} \stackrel{\text{def}}{=} \forall pkt, \texttt{goodPkt}(pkt) \Rightarrow \mathsf{WF}(\texttt{dlvStep}_{pkt})$$

where $\texttt{goodPkt}(pkt)$ states that pkt is sent between honest nodes:

$$\texttt{goodPkt}(pkt) \stackrel{\text{def}}{=} \texttt{isByz}\ pkt.\texttt{src} = \texttt{false} \wedge \texttt{isByz}\ pkt.\texttt{dst} = \texttt{false}$$

In this definition, WF is the *weak fairness* combinator [36] and $\texttt{dlvStep}_{pkt}$ is a TLA action stating that the system takes a Delivery step tagged with $\mathsf{D}(pkt)$, given that pkt is fresh in the packet soup:

$$\texttt{dlvStep}_{pkt}(w, w') \stackrel{\text{def}}{=} pkt \in w.\texttt{packetSoup} \wedge pkt.\texttt{received} = \texttt{false} \Rightarrow w \xrightarrow{\mathsf{D}(pkt)} w'$$

To justify the definition of `WFDelivery` given above, we prove that for any trace σ such that $\sigma \vDash \Box\langle \texttt{next} \rangle$, $\sigma \vDash \texttt{WFDelivery}$ if and only

if the following predicate holds on σ:

$$\texttt{eventualDelivery}(\sigma) \stackrel{\text{def}}{=}$$
$$\forall pkt, n \in \mathbb{N}, \texttt{goodPkt}(pkt) \land pkt \in \sigma(n).\texttt{packetSoup} \land$$
$$pkt.\texttt{received} = \texttt{false} \Rightarrow \exists k \in \mathbb{N}, \sigma(n+k) \xrightarrow{\mathrm{D}(pkt)} \sigma(n+k+1)$$

This predicate can be viewed as a "finer-grained" version of the eventual delivery assumption, stating that every packet sent between honest nodes at timestamp n will eventually be delivered at timestamp $n+k$, where k is an unknown natural number but is guaranteed to exist.

Apart from the symbols explained above, in CoqTLA, an entailment $P \vdash Q$ holds if for any σ, $\sigma \vDash P$ implies $\sigma \vDash Q$. It allows for liveness specifications to be expressed in the form of

$$\ulcorner \texttt{init} \urcorner \land \Box\langle \texttt{next} \rangle \land \texttt{WFDelivery} \vdash \ulcorner P_b \urcorner \leadsto \ulcorner P_e \urcorner$$

where the *leads-to* connective $P \leadsto Q$ is defined as $\Box(P \Rightarrow \Diamond Q)$ in LTL ($\Diamond$ is the *eventual* operator). This entailment can be interpreted as: during the execution of a system, under the eventual delivery assumption (equivalent to `WFDelivery`), whenever the system enters the state described by P_b, it is guaranteed to eventually reach the state described by P_e. The entailment in Fig. 4 is then obtained by properly instantiating the P_b and P_e above.

One important lemma involved in translating the informal proof from Sec. 2.3.3 into formal proof scripts is the *transitivity* of $\leadsto$:

$$\forall P, Q, R,\ P \leadsto Q \land Q \leadsto R \vdash P \leadsto R \qquad \text{(LeadstoTrans)}$$

This lemma is indispensable for achieving modular liveness proofs. Recall that in the informal proof, the protocol is divided into the "initial" phase and the "echo" phase. By allowing us to reason about these two phases independently (*i.e.*, show $P \leadsto Q$ and $Q \leadsto R$ separately), this lemma bridges these phases and completes the final step towards the desired termination property (*i.e.*, $P \leadsto R$).

3.3 A Functor for Protocol Composition

As the final ingredient of Bythos, we describe one of its key innovations: a functor for combining two verified BFT protocol instances, along with their safety and liveness. Our construction is motivated by Abraham *et al.*'s observation [1–3] that sequentially executing the logic of the same or different BFT protocols may help to increase the amount of knowledge that a node has about the preceding execution of the system and the amount of trust in other nodes. To reflect this observation, Bythos provides a mechanism for *sequential composition* of protocol instances, such that when executing the protocol composed of P_A and P_B, an honest node will execute the actions of the protocol P_A first, and upon reaching a certain stage (for example, when P_A outputs a value), it starts the execution of another protocol, P_B. This sequential composition can be iterated.

Specifically, we allow users to sequentially compose protocols implemented in Bythos by providing a functor that takes two protocol instances P_A, P_B, defined by following the steps from Sec. 3.1.1–3.1.3, as inputs and provides the construction of the composite protocol in which each honest node runs the logic of P_A and P_B sequentially. To determine the starting conditions for P_B, the user only needs to specify two *triggers*, which determine when a node should start to run P_B, based on the node's local states before and after executing the `procInt` or `procMsg` of P_A:

```
Parameter trigger_procMsg :
  P_A.State (* local state before executing P_A.procMsg *) ->
  P_A.State (* local state after executing P_A.procMsg *) ->
  option P_B.InternalEvent.
Parameter trigger_procInt : (* the same type as above *)
```

The functor then constructs the composite protocol by setting its local state to be the pair of those of P_A and P_B, and making its message type be the *sum type* of those of P_A and P_B. The `procMsg` handler of the composite protocol works by checking whether the incoming message is for P_A or P_B and calling the `procMsg` of the corresponding protocol accordingly, whereas the `procInt` of the composite protocol only handles the internal events of P_A. When the triggers of these two handlers instruct that P_B should also start executing, they will additionally call P_B's `procInt`.

We note that the composite protocol instance produced by the functor is on its own a protocol which can be plugged back into the functor, thereby enabling iterated sequential composition. Moreover, the system instance of the composite protocol is obtained by following the same steps as other protocols (Sec. 3.1.4–3.1.6), with its `byzConstraints` defined simply using those of P_A and P_B. Consequently, the utilities for reasoning about liveness introduced in Sec. 3.2 are also applicable to the composite protocol.

The composition functor also provides relevant lemmas that "lift" the safety and liveness properties of sub-protocols to the composite protocol. This lifting is feasible because each transition step of the composite protocol can be "projected" into a valid transition step of each sub-protocol, facilitated by the presence of Stuttering steps. For example, an Internal step of the composite protocol where the trigger is not enabled can be projected into an Internal step of P_A and a Stuttering step of P_B (or an Internal step of P_B if the trigger is enabled instead). In this way, execution traces of the composite protocol can be projected into those of sub-protocols, ensuring that their safety and liveness properties are preserved by the composite protocol's execution. Thanks to the transitivity of the leads-to connective (LeadstoTrans), we can even *compose the liveness properties* of P_A and P_B to derive liveness properties of the composite protocol which cannot be derived from any of the individual sub-protocols in isolation (*cf.* Sec. 4.3).

3.4 Extracting an Executable Implementation

The definitions of the protocol components outlined in Sec. 3.1.1–3.1.3 are sufficient to derive executable reference implementations. We use the standard mechanism of extracting OCaml code from Coq [40], which turns Coq data types into corresponding ones in OCaml, and converts handler functions `procInt` and `procMsg` into the respective OCaml definitions. No additional effort is required to extract a composite protocol, since the extraction works on protocol instances, and composite protocols are also "first-class" protocol instances (*cf.* Sec. 3.3). The working reference implementation is obtained by linking the extracted protocol code to the trusted (*i.e.*, unverified) network shim [48], which is assumed to correctly capture the network behaviour as per the semantics from Sec. 3.1.6, as well as the trusted concrete implementations of the cryptographic primitives used by the protocol's handlers.[2]

[2]In our artefact [66], the threshold signature scheme is, for simplicity, implemented assuming public-key infrastructure, avoiding the need for key distribution.

Algorithm 1 Reliable Broadcast (for node p)

```
1:  action RBcast(Round r, Value v):
2:      send Init(r, v) to all nodes
3:  upon receiving Init(r, v) from node q:
4:      if p has never sent Echo(q, r, v') before for any v':
5:          send Echo(q, r, v) to all nodes
6:  upon receiving Echo(q, r, v) from ≥ n − f different senders
7:    or receiving Vote(q, r, v) from ≥ f + 1 different senders:
8:      if p has never sent Vote(q, r, v') before for any v':
9:          send Vote(q, r, v) to all nodes
10: upon receiving Vote(q, r, v) from ≥ n − f different senders:
11:     output (q, r, v)
```

4 More Case Studies

In this section, we showcase more of the distributed protocols we have verified in Bythos *wrt.* their safety and liveness specifications.

4.1 Reliable Broadcast

We begin with Bracha's Reliable Broadcast (RB) [13, 14], a classic BFT broadcast protocol in the asynchronous setting with at most f adversaries and at least $2f + 1$ honest nodes. The protocol assumes the existence of authenticated channels, no message loss or tampering, and no public-key infrastructure (PKI), *i.e.*, no cryptographic signatures. We consider the multi-shot version of this protocol [9, 44], shown in Algorithm 1. Similar to the Provable Broadcast in Sec. 2, this variant allows an arbitrary sender p to broadcast different values in distinct *rounds*. All messages are implicitly decorated with the sender's identity, provided by the authenticated channel. The protocol has three types of messages: Init, Echo and Vote. A node p *reliably broadcasts* a value v at round r by sending $\mathsf{Init}(r, v)$ to all nodes (line 2). The Echo and Vote messages are for securing the broadcast in the presence of the Byzantine adversary. Once a node sees $n - f$ votes for the same value in a round, it *outputs* that value, acknowledging its receipt from q (line 11).

Reliable Broadcast is expected to satisfy the following specification comprising two safety and two liveness properties:

- *Integrity* (S): For any two honest nodes p and q, if p outputs (q, r, v), then q must have reliably broadcast v at round r.
- *Agreement* (S): For any two honest nodes p_1 and p_2, if p_1 outputs (q, r, v_1) and p_2 outputs (q, r, v_2), then $v_1 = v_2$.
- *Global liveness* (L): If an honest node p outputs (q, r, v), then every honest node will eventually output (q, r, v).
- *Validity* (L): If an honest node q reliably broadcast v at round r, then every honest node will eventually output (q, r, v).

Intuitively, the agreement and validity properties mean that RB can serve as a reliable broadcast primitive for honest (*i.e.*, non-faulty) nodes. Global liveness guarantees that all honest nodes eventually have the same behaviour even if the sender is Byzantine: they either all output the same value, or none output at all.

We encoded Reliable Broadcast in Bythos, in the same manner as Provable Broadcast. Specifically, a node's local state is defined as a `Record`, with each field capturing information such as the values the node has echoed, voted for or output, the list of nodes from which it has received certain Echo or Vote messages, and so on.

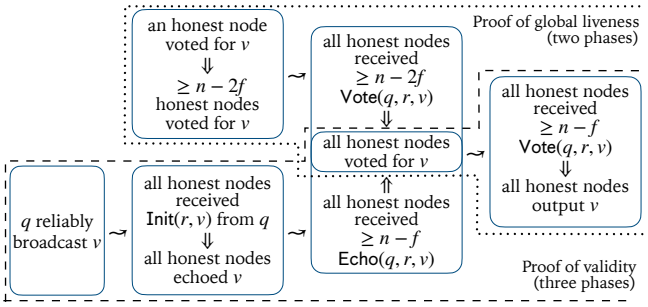

Fig. 5: Phase decomposition for the liveness proofs of Reliable Broadcast. A box represents the start or end of a phase, and a ⇝ symbol indicates the proof goal that a phase progresses from start to end. The area enclosed by dotted lines corresponds to the proof of global liveness, while the area enclosed by dashed lines corresponds to the proof of validity.

The message type is an algebraic data type with three constructors corresponding to Init, Echo, and Vote respectively. The handlers are straightforward translations from the pseudocode (`procInt` corresponds to RBcast at line 1, while **upon** checks are performed in `procMsg`). Since no cryptographic primitive is involved in RB, `byzConstraints` is set to `(fun _ _ => True)`.

4.1.1 Knowledge-Driven Proofs of Safety. We prove the safety properties in the same fashion as Sec. 2.2.5, incrementally establishing an inductive invariant that implies both safety properties by devising knowledge lemmas of different categories. For RB, the knowledge lemmas in the categories (1)–(4) in Sec. 2.2.5 closely resemble those used in the safety proof of PB. As an example, the data persistence properties in RB can be summarised as: a field in the local state of an honest node is never overwritten or evolves monotonically (*e.g.*, set of received messages only grows). Therefore, for brevity, we omit the statements of those knowledge lemmas and instead only list representative implications of knowledge as follows (fixing q as the broadcast initiator and r as the round):

(1) *Echo before vote:* if an honest node has voted for v, there is at least one honest node that echoed v;
(2) *Vote integrity:* if an honest node voted for v and q is honest, q did reliably broadcast v at round r;
(3) *Integrity:* if an honest node output a value v and q is honest, q did reliably broadcast v at round r;
(4) *Agreement:* if two honest nodes output, they must output the same value, regardless of whether q is honest.

The properties (3) and (4) are exactly RB's two safety properties.

4.1.2 Knowledge-Driven Proofs of Liveness. We prove liveness in the same fashion as Sec. 2.3.3. For each liveness property, we first decompose the protocol execution into several phases, so that the liveness property can be formulated as "the start of the first phase leads to the end of the last phase".

For conciseness, we only depict the phase decomposition in Fig. 5 without going into details. As we have done in Sec. 2.3.3, at the start of each phase, we use local-to-packet knowledge lemmas to infer which packets appear in the packet soup. Based on the fair delivery assumption, we know that they will eventually be received.

We then apply knowledge lemmas–especially the packet-to-local receipt-sensitive ones–for reasoning at the end of the current phase (represented by short arrows in Fig. 5). After dealing with all phases, we finally bridge them using (Leadsto-Trans). Remarkably, both global liveness and validity share the same last phase. Thanks to the proof modularity via phase decomposition and (Leadsto-Trans), the proof for this last phase is reused across both properties.

4.2 Accountable Byzantine Confirmer

Our next case study is the Accountable Confirmer (AC) protocol [22], which can be attached to an arbitrary non-synchronous BFT consensus protocol, via the Accountable Byzantine Consensus ($\mathcal{ABC}$) transformation, to add *accountability* when the number of Byzantine nodes f' *exceeds* the threshold f. AC requires both threshold signature scheme and public-key infrastructure; to avoid ambiguity, we refer to the PKI-based signatures as *standard* ones. Given a BFT consensus protocol that satisfies the standard agreement, validity, and termination properties (statement omitted), $\mathcal{ABC}$ produces an *accountable BFT consensus protocol* that still satisfies those properties when $f' \leq f$, and additionally satisfies the *accountability* (L): if two honest nodes decide different values, then every honest node eventually detects at least $n - 2f$ Byzantine nodes and obtains a proof of their culpability. The accountability, essentially provided by AC, says that *if* the Byzantine adversary has compromised the safety of the consensus (which, for a correct protocol, can only happen if $f' > f$), then every honest node will eventually detect this, and moreover, identify the culprits *and* produce a cryptographically verifiable proof of their misbehaviour.

It is surprising that accountability can be added to an *arbitrary* underlying protocol–regardless of the concrete value of f–without any knowledge of its internals, and moreover, without increasing overall communication complexity in the normal case. The intuition behind AC (outlined in Algorithm 2) and the $\mathcal{ABC}$ transformation, however, is simple. Basically, the accountable protocol produced by $\mathcal{ABC}$ can be regarded as the sequential composition of its underlying consensus protocol and AC. Whenever a node *decides* a value v in the underlying consensus, the node *submits* v in AC (line 3) by broadcasting v along with a partial signature and a standard signature for v (line 5). When a node receives a Submit message containing the node's submitted value and valid signatures, it will record the sender and the signatures (lines 10–11). Once the number of collected senders reaches $n - f$, a node *confirms* its submitted value (a confirmation in AC is the decision in the transformed accountable protocol), combines the previously recorded partial signatures into a *light certificate* and broadcasts it (lines 12–14). If a confirmed node sees light certificates for conflicting values—which suggests that safety might be violated in the underlying consensus—it broadcasts a *full certificate* consisting of all recorded senders and standard signatures from them (lines 15–18). Since the threshold of confirmation is $n - f$, two full certificates for conflicting values must intersect in at least $n - 2f$ nodes which signed different values, and the associated standard signatures can be used as positive proof of misbehaviour, thus providing *accountability* (line 22).

We note that two different signature schemes are used here to control communication complexity and accommodate different axioms on cryptographic primitives. Interested readers can find the full details in the original paper [22].

Algorithm 2 Accountable Confirmer (for node p)

1: **on initialisation**:
2: set `buffer`, `fromset`, `psset`, `nsset`, `lcertset`, `certset` to $\emptyset$; set `acval` to $\perp$; set `confirmed` to `false`
3: **action** ACSubmit(Value v): ▷ called only once
4: `acval` := v
5: **send** Submit(v, $partial\_sign_p(v)$, $sign_p(v)$) to all nodes
6: **for each** Submit message *msg* **in** `buffer`:
7: process *msg* according to line 8
8: **upon receiving** Submit(v, ps, s) from node q:
9: **if** `acval` is $\perp$: add this Submit message to `buffer`
10: **else if** ps and s can be verified *wrt.* v with q's public key, `acval` = v, `confirmed` = `false` and $q \notin$ `fromset`:
11: add q to `fromset`; add ps to `psset`; add (q, s) to `nsset`
12: **upon** |`fromset`| $\geq n - f$:
13: `confirmed` := `true`
14: **send** LightCert(`acval`, *combine*(`psset`)) to all nodes
15: **upon receiving** LightCert(v, cs):
16: **if** cs can be verified *wrt.* v: add (v, cs) to `lcertset`
17: **upon** `confirmed` = `true` and $\exists (v, cs), (v', cs') \in$ `lcertset`, $v \neq v'$:
18: **send** Cert(v, `nsset`) to all nodes
19: **upon receiving** Cert(v, nss):
20: **if** $|nss| \geq n - f$ and for any $(q, s) \in nss$, s can be verified *wrt.* v with q's public key: add (v, nss) to `certset`
21: **upon** $\exists (v, nss), (v', nss') \in$ `certset`, $v \neq v'$:
22: **detect** $\{q : \exists s, s', (q, s) \in nss \wedge (q, s') \in nss'\}$

4.2.1 Uncovering Implicit Assumptions. In the process of formalising AC, we discovered an implicit assumption required for the protocol to be live. Concretely, to ensure that the transformed accountable protocol satisfies termination and agreement, AC has to satisfy *terminating convergence*, a liveness property: if $f' \leq f$ and all honest nodes submit the same value v, then v is eventually confirmed by every honest node. In the original formulation of AC (*i.e.*, Algorithm 2 excluding grey parts), this property *may not hold* due to asynchrony, even if we assume fair delivery. To see that, consider a "slow" node that submits only after all the other nodes have submitted. Before it submits, when receiving Submit messages from other nodes, the slow node would do nothing due to the check on line 10. Consequently, the poor slow node may not have a chance to reach the required $n - f$ threshold and confirm, since it may not receive enough Submit messages after it submits.

To address this issue, nodes that have not yet submitted must *buffer* (*i.e.*, store) received Submit messages until they submit and can thus examine the messages to discern whether to accept them or not. This buffering introduces extra complexity in both the implementation and knowledge lemma proofs, requiring that the `procInt` handler that processes the ACSubmit event (line 3) call `procMsg` in a loop over all received and buffered messages.

4.2.2 Liveness Proof. With the buffering in place, the liveness proof proceeds similarly to that for Reliable Broadcast, by a decomposition into phases. Terminating convergence is proven in a single

phase: the "submit" phase starts once all honest nodes submit the same value, and hence broadcast Submit messages with signatures, and ends when these messages are received by all honest nodes. At that point, every honest node will confirm since $n - f' \geq n - f$. Accountability is proven in two phases: (1) a "confirm" phase that starts once two honest nodes confirmed different values and hence broadcast conflicting light certificates, and (2) a "detect" phase that starts once the two honest nodes received conflicting light certificates and hence broadcast their full certificates. Once the second phase ends, all honest nodes will have received two conflicting full certificates and be able to detect at least $n - 2f$ culprits (line 22).

4.3 Accountable Reliable Broadcast

The Accountable Byzantine Consensus paper [22] also defines a generalised $\mathcal{ABC}$ transformation that can be applied to any agreement protocol, including Reliable Broadcast. In this section, we demonstrate our composition of the Accountable Confirmer from Sec. 4.2 with the Reliable Broadcast instance we formalised in Sec. 4.1. The definition of the novel composed protocol, which we call ARB, is straightforward conceptually. The formalisation of the composition is more involved, as we have already detailed in Sec. 3.3. The connection between RB and AC is done by `trigger_procMsg` (*cf.* Sec. 3.3), which fires the ACSubmit event in AC once RB outputs.

4.3.1 Compositional Liveness Proof. Based on the *validity* of RB (*cf.* Sec. 4.1) and the *terminating convergence* of AC (*cf.* Sec. 4.2.1), we want to prove the *overall validity* of the ARB composition, which states that if an honest node q reliably broadcast v, then v is eventually confirmed by every honest node, under the assumption that the threshold f is not breached. This proof follows the same technique of decomposition into phases described earlier, except now the phases are the two different protocols, RB and AC. Similar to the previous proofs, the most involved part is showing that the end of one phase (in this case, when all honest nodes output a value v in RB) corresponds to the beginning of the next phase (all honest nodes in AC submitted v). We prove this correspondence by propagating knowledge across the protocol boundary via specialised invariants, which follow from the implementation of `trigger_procMsg` and the invariants of RB and AC. One of those invariants in particular, which we refer to as the *connector*, ensures that AC will proceed as expected following RB; it states that if an honest node output a value v in RB, then it submitted v in AC. By reusing the liveness proofs of RB validity and AC terminating convergence, and integrating them with the connector and (Leadsto-Trans), the Coq proof of overall validity is condensed to merely 7 lines.

5 Discussion

In this section, we discuss both the quantitative and qualitative aspects of our verification work from a user's perspective.

Proof effort. We summarise our verification efforts in Tab. 1. Notably, the number of **Proof** lines in protocol implementation may be non-zero, since some definitions are constructed using tactics in the proof mode, and we might reason about some definitions as soon as defining them. For example, the implementation of Accountable Confirmer requires a *detecting* function (*cf.* line 22 in Algorithm 2), which, given a list of full certificates, returns all culprits within them. To ensure this function's correctness, after implementing it, we immediately prove that its implementation satisfies a certain specification, which is later used in the proof of accountability.

Table 1: Statistics of the formal development, including Bythos and all formalised protocols, in lines of Coq code.

Library	Component	Spec	Proof	Total
Bythos (Sec. 3)	System (Sec. 3.1)	729	465	1194
	Liveness (Sec. 3.2)	160	181	341
	Composition (Sec. 3.3)	329	255	584
	Utilities	184	157	341
	Total	1402	1058	2460
Provable Broadcast (Sec. 2)	Implementation (Sec. 2.1)	121	6	127
	Safety (Sec. 2.2)	404	320	724
	Liveness (Sec. 2.3)	92	67	159
	Composition (Sec. 2.4)	85	10†	95
	Total	702	403	1105
Reliable Broadcast (Sec. 4.1)	Implementation	130	6	136
	Safety (Sec. 4.1.1)	448	432	880
	Liveness (Sec. 4.1.2)	144	161	305
	Total	722	599	1321
Accountable Confirmer (Sec. 4.2)	Implementation	237	109	346
	Safety	619	709	1328
	Liveness (Sec. 4.2.2)	172	200	372
	Total	1028	1018	2046
Accountable Reliable Broadcast (Sec. 4.3)	Implementation	33	0	33
	Connector (Sec. 4.3.1)	48	92	140
	Liveness (Sec. 4.3.1)	3	7	10
	Total	84	99	183

† For both twice-iterated and thrice-iterated PB, each one is 5 lines.

The total size of our Coq development is around 7100 lines of code. The trusted shim layer in OCaml (*cf.* Sec. 3.4), not included into the statistics, contains around 200 lines of code.

Both the Provable Broadcast and Reliable Broadcast require roughly 1000 lines of Coq proof each. The proof of Accountable Confirmer, however, is almost twice as long. The fundamental reason for this is that, because of the buffering (explained in Sec. 4.2.1), `procInt` may invoke `procMsg` an indefinite number of times (determined by the size of buffer). Therefore, in the safety proof, we must first prove that the inductive invariant is preserved under any Delivery step, and then do the same for any Internal step. This two-step process leads to the inflation of the proof size.

Limitations. The major limitation of Bythos is its *scalability*, in the sense that it might require substantial human efforts to verify a complex Byzantine protocol in Bythos. When verifying the protocols listed in Tab. 1, all knowledge lemmas were discovered manually through trial and error, and the proofs showing that they form inductive invariants were also manually created (by typing proof scripts). Each of these protocols has 10 to 20 knowledge lemmas; accordingly, for more complex protocols, our knowledge-driven proof methodology will necessitate a more challenging knowledge discovery process and longer proofs. Moreover, some protocols may involve advanced features such as probabilistic termination [4], which Bythos is unable to help reason about.

6 Related Work

Computer-aided verification of distributed protocols, both interactive and automated, has been a very active research area for the last decade. Tab. 2 summarises the related approaches and frameworks.

Frameworks for foundational verification. The seminal work on IronFleet [30] has presented the first framework for deductive verification of both safety and liveness properties of real-world executable distributed systems. IronFleet is implemented on top of the deductive Dafny verifier [38], which was also used for embedding the specifications in the style of Lamport's TLA [36] and the respective rules of Linear Temporal Logic (LTL), similarly to the TLA+ embedding into Coq employed by Bythos. IronFleet did not encode Byzantine network semantics and, thus, did not allow for verification of BFT protocols in the respective environment. It also did not consider verification of composite protocols. While the logic of IronFleet and the semantics of protocol verified in it has been encoded in terms of Dafny language, Dafny itself relies on an Z3 SMT solver [23] to discharge the verification conditions (VCs). Therefore, verification in IronFleet can be considered foundational with the reservation that one trusts the correctness of Dafny VC generator and the underlying SMT solver (hence $\checkmark_{\text{SMT}}$ in Tab. 2).

Developed concurrently with IronFleet, Verdi [63] is a foundational framework embedded into Coq aiming to reduce the trusted code base down to the implementation of Coq proof checker. Used to verify the safety specification of Raft consensus protocol [64], Verdi allows for a limited form of proof composition with the mechanism of *verified system transformers* that allow for strengthening assumptions about (non-Byzantine) network semantics.

Toychain [50] was the first attempt to formalise the safety of a BFT blockchain protocol in a foundational proof assistant (also Coq), executable via code extraction to OCaml, although its network semantics did not account for the Byzantine behaviours. Thanks to its simplicity, the Toychain approach to formulate the semantics of blockchain protocols has inspired a number of follow-up safety verification efforts on Algorand [7], Casper [47], HotStuff [18], as well as a proof of safety and liveness of Nakamoto Proof of Stake blockchain consensus in a synchronous network [61].

Disel [57] was the first framework to tackle compositional foundational verification of composite protocols by implementing a version of Separation Logic [56] and allowing its logical assertions to quantify over the network state spanning different sub-protocol instances. Disel's verification capabilities have been later enhanced in the works on Aneris [34] and Grove [58], both implemented on top of Coq-based Iris separation logic embedding [31] and allowing for node-local concurrency and crash recovery, respectively. Neither Aneris nor Grove support reasoning about Byzantine systems, nor do they allow to verify liveness properties.

TLC (Temporal Logic of Components) [29] is a Coq-based framework that features a temporal program logic and offers inference rules to reason about safety and liveness of vertically-composed distributed system stacks. We conjecture that TLC should be able to accommodate proofs about horizontally composed protocols as well, yet the TLC paper does not showcase it in that capacity. TLC does not support proofs about Byzantine fault tolerance.

Velisarios [54] and Asphalion [62] are two verification frameworks embedded into Coq that aim specifically at reasoning about safety specification of BFT consensus protocols and are based on Logic of Events [12]. Stating properties and reasoning in the Logic of Events is quite different in flavour from the state-based inductive proofs conducted in Bythos. That said, the knowledge lemmas we phrase in our proofs are reminiscent to the rules of the knowledge calculus employed by Asphalion. Both Velisarios and Asphalion leave reasoning about liveness properties for the future work.

Finally, LiDO [53] is a recent verification framework, which allows one to verify BFT protocols for both safety and liveness, yet whose take on composition is quite different from ours. Specifically, LiDO achieves *vertical* composition via refinement, allowing one to verify validity of optimisations within the *same* protocol. In contrast, Bythos allows for *horizontal* composition, *i.e.*, reusing proofs when assembling multi-stage protocols from basic ones.

Table 2: Comparison with existing frameworks for machine-assisted verification of distributed protocols on the grounds of supporting proofs of *Safe*ty, *Live*ness, *Byz*antine network, allowing for horizontal protocol *Comp*osition, being *Found*ational, and capable of generating *Exec*utable code.

Framework		*Safe*	*Live*	*Byz*	*Comp*	*Found*	*Exec*
IronFleet [30]	2015	✓	$\checkmark_{\text{P}}$			$\checkmark_{\text{SMT}}$	✓
Verdi [63]	2015	✓				✓	✓
PSync [26]	2016	✓	$\checkmark_{\text{P}}$				✓
Ivy [46]	2016	✓					
Toychain [50]	2018	✓				✓	✓
Disel [57]	2018	✓			✓	✓	✓
Padon *et al.* [45]	2018	✓	$\checkmark_{\text{A}}$				
Taube *et al.* [59]	2018	✓					✓
Velisarios [54]	2018	✓		✓		✓	✓
Asphalion [62]	2019	✓		✓	✓	✓	✓
Aneris [34]	2020	✓			✓	✓	✓
TLC [29]	2020	✓	$\checkmark_{\text{P}}$		✓	✓	
Losa & Dodds [42]	2020	✓	$\checkmark_{\text{A}}$	✓			
Thomsen & Spitters [61]	2021	✓	$\checkmark_{\text{S}}$	✓		✓	
Carr *et al.* [18]	2022	✓		✓		✓	
ByMC [32]	2023	✓	$\checkmark_{\text{A}}$	✓	✓		
Grove [58]	2023	✓			✓	✓	✓
LiDO [53]	2024	✓	$\checkmark_{\text{P}}$	✓		✓	✓
Bythos (this work)	2024	✓	$\checkmark_{\text{A}}$	✓	✓	✓	✓

Tools for automated verification. Concurrently with foundational verification frameworks, a number of domain-specific tools for distributed protocol verification have been proposed, each exploiting certain assumptions about the systems being verified. Those approaches typically sacrifice foundational guarantees, axiomatising the semantics of their programming language, as well as the ability to compose specifications, for the sake of proof automation.

PSync [26] is a domain-specific language based on the *heard-of model* [21], which allows the user to implement and automatically verify their protocols that assume round-based communication in a partially synchronous model. It supports checking of both safety and liveness properties that are expressible in a specialised fragment of first-order logic and can be proved using SMT solvers [25].

Ivy [46] is an automated verification tool that expects the protocols, their specifications, and the inductive invariants provided by the user as quantified formulas to produce verification conditions

that fit into a specific decidable fragment of the first-order logic [51]. Ivy has been later demonstrated to be able to encode and prove liveness specifications by reducing them to safety properties [45]; it was also extended to produce executable C++ code from the model definitions [59] and used to verify both safety and liveness of a simplified Stellar Consensus Protocol [42]. Due to the first-order nature of the Ivy logic, it does not allow for specification composition and reuse of proofs about independently verified protocols.

ByMC [32] is a symbolic model checker that automates safety and liveness proofs for BFT protocols [11]. The proof is done via the decomposition similar to what is offered by Bythos. That said, for the verification, ByMC relies on an unverified translation of an protocol's pseudo-code to a *threshold automaton*. Similarly to other automated tools, ByMC imposes restrictions on the shapes of properties it can express, requiring them to be for the form $\overline{\forall R \in \mathbb{N}}, \phi[\overline{R}]$, where ϕ is a quantifier-free formula in LTL. It means, in particular, that ByMC would not be able to directly encode and prove the property (1) of Reliable Broadcast from Sec. 4.1.1.

Liveness and synchrony. We conclude with a brief discussion on the nature of liveness proofs supported by approaches from Tab. 2.

Due to the famous FLP impossibility result [28], one would need to make certain assumptions about synchrony in the network to establish a desired termination property of any *fault-tolerant consensus* protocol. To wit, the work by Thomsen and Spitters proves liveness of Nakamoto-style blockchain consensus assuming *strong synchrony* [61], where an upper bound on the time it takes to deliver a message is statically determined—hence the mark $\checkmark_S$ in Tab. 2.

With the same rationale, IronFleet [30], PSync [26], TLC [29], and LiDO [53] assume *partially* synchronous network ($\checkmark_P$) [27], in which the message delivery latency has a fixed bound that is guaranteed to hold after certain unspecified moment *a.k.a. global synchronisation time* (GST). Each of those frameworks encodes partial synchrony differently: IronFleet imposes additional assumptions on the considered executions of the specific protocol in question [30, §5.1.4]; PSync adds the GST existence to the axiomatisation of the network semantics in the first-order logic [26, §5.3]; TLC features an explicit logical rule postulating the existence of GST [29, Fig. 11]; LiDO's handling of GST is done semantically at the level of system traces [53, §2.4], similarly to that of PSync. Despite being the closest in its expressive power to Bythos, LiDO does not offer a *proof system* (*i.e.*, logic) for liveness proofs (*e.g.*, by means of TLA-style rules): all its liveness proofs are in terms of explicit execution traces.

Similarly to Ivy [42, 45] and ByMC [11], Bythos is geared towards proofs of *eventual* liveness in an *asynchronous* network under a fairness assumption ($\checkmark_A$), which is sufficient to state many useful specifications (as we have shown in Sec. 2 and Sec. 4)—though some problems (like deterministic fault-tolerant consensus), are impossible in an asynchronous network. Adopting the weakest form of synchrony in Bythos has allowed for very short liveness proofs via TLA-style logic rules (as compared to much more intricate formalism of TLC [29]), as well as compositional liveness verification. Moreover, proving protocol liveness in asynchrony immediately implies protocol liveness in synchrony and partial synchrony. That said, we recognise the advantage of supporting partial synchrony assumptions, and consider it our future work.

7 Conclusion

The Bythos framework, together with its methodology, streamlines the verification of Byzantine Fault-Tolerant protocols and their compositions. Bythos allows users to encode protocols in a way closely aligned with their informal specifications in pseudocode and to reason about them using a standard toolset of invariant-based safety reasoning and TLA-based liveness proofs. Through systematic decomposition, correctness proofs, especially the liveness ones, can be reused within a standalone protocol or across components of a composite protocol. Beyond these aspects, our novel knowledge-driven proof methodology clarifies causal relations within a protocol and structures its correctness proof in an incremental fashion, thereby rendering the proof more tractable.

Acknowledgments

We thank Ittai Abraham for his comments. We also thank the anonymous CCS'24 reviewers for their feedback. This work was partially supported by a Singapore Ministry of Education (MoE) Tier 3 grant "Automated Program Repair" MOE-MOET32021-0001, MoE Tier 1 grant T1 251RES2108 "Automated Proof Evolution for Verified Software Systems", and by Sui Academic Research Award.

Data Availability

The artefact with a snapshot of the Coq development accompanying this paper is available online [66]. It contains Bythos source code and the implementation of all case studies from Sec. 2 and Sec. 4.

References

[1] Ittai Abraham. 2022. Linear PBFT: a gentle introduction to Practical Byzantine Fault Tolerance. https://decentralizedthoughts.github.io/2022-11-20-pbft-via-locked-braodcast/
[2] Ittai Abraham. 2022. Provable Broadcast. https://decentralizedthoughts.github.io/2022-09-10-provable-broadcast/
[3] Ittai Abraham. 2022. Two Round HotStuff. https://decentralizedthoughts.github.io/2022-11-24-two-round-HS/
[4] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. 2019. Asymptotically Optimal Validated Asynchronous Byzantine Agreement. In *PODC*. ACM. https://doi.org/10.1145/3293611.3331612
[5] Danel Ahman, Cédric Fournet, Catalin Hritcu, Kenji Maillard, Aseem Rastogi, and Nikhil Swamy. 2018. Recalling a witness: foundations and applications of monotonic state. *Proc. ACM Program. Lang.* 2, POPL (2018). https://doi.org/10.1145/3158153
[6] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. 2018. Chainspace: A Sharded Smart Contracts Platform. In *NDSS*. The Internet Society. https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_09-2_Al-Bassam_paper.pdf
[7] Musab A. Alturki, Jing Chen, Victor Luchangco, Brandon M. Moore, Karl Palmskog, Lucas Peña, and Grigore Rosu. 2019. Towards a Verified Model of the Algorand Consensus Protocol in Coq. In *FM Workshops (LNCS, Vol. 12232)*. Springer. https://doi.org/10.1007/978-3-030-54994-7_27
[8] Andrew W Appel. 2001. Foundational Proof-Carrying Code. In *LICS*. IEEE Computer Society. https://doi.org/10.1109/LICS.2001.932501
[9] Hagit Attiya and Jennifer L. Welch. 2004. *Distributed computing - fundamentals, simulations, and advanced topics (2. ed.)*. Wiley.
[10] Idan Berkovits, Marijana Lazic, Giuliano Losa, Oded Padon, and Sharon Shoham. 2019. Verification of Threshold-Based Distributed Algorithms by Decomposition to Decidable Logics. In *CAV (LNCS, Vol. 11562)*. Springer. https://doi.org/10.1007/978-3-030-25543-5_15
[11] Nathalie Bertrand, Vincent Gramoli, Igor Konnov, Marijana Lazic, Pierre Tholoniat, and Josef Widder. 2022. Holistic Verification of Blockchain Consensus. In *DISC (LIPIcs, Vol. 246)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPICS.DISC.2022.10
[12] Mark Bickford, Robert Constable, and Vincent Rahli. 2012. The Logic of Events, a framework to reason about distributed systems. In *Languages for Distributed Algorithms (LADA) workshop*.

[13] Gabriel Bracha. 1984. An Asynchronou [(n-1)/3]-Resilient Consensus Protocol. In *PODC*. ACM. https://doi.org/10.1145/800222.806743

[14] Gabriel Bracha. 1987. Asynchronous Byzantine Agreement Protocols. *Inf. Comput.* 75, 2 (1987). https://doi.org/10.1016/0890-5401(87)90054-X

[15] Ethan Buchman. 2016. *Tendermint: Byzantine Fault Tolerance in the Age of Blockchains.* Master's thesis. University of Guelph.

[16] Ethan Buchman, Jae Kwon, and Zarko Milosevic. 2018. The latest gossip on BFT consensus. *CoRR* abs/1807.04938 (2018). arXiv:1807.04938 http://arxiv.org/abs/1807.04938

[17] Vitalik Buterin and Virgil Griffith. 2017. Casper the Friendly Finality Gadget. *CoRR* abs/1710.09437 (2017). http://arxiv.org/abs/1710.09437

[18] Harold Carr, Christa Jenkins, Mark Moir, Victor Cacciari Miraldo, and Lisandra Silva. 2022. Towards Formal Verification of HotStuff-Based Byzantine Fault Tolerant Consensus in Agda. In *NASA Formal Methods (LNCS, Vol. 13260)*. Springer. https://doi.org/10.1007/978-3-031-06773-0_33

[19] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *OSDI*. USENIX Association. https://dl.acm.org/citation.cfm?id=296824

[20] Tej Chajed. 2024. TLA in Coq. https://github.com/tchajed/coq-tla.

[21] Bernadette Charron-Bost and André Schiper. 2009. The Heard-Of model: computing in distributed systems with benign faults. *Distributed Comput.* 22, 1 (2009). https://doi.org/10.1007/S00446-009-0084-6

[22] Pierre Civit, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, and Jovan Komatovic. 2022. As easy as ABC: Optimal (A)ccountable (B)yzantine (C)onsensus is easy!. In *IPDPS*. IEEE. https://doi.org/10.1109/IPDPS53621.2022.00061

[23] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (LNCS, Vol. 4963)*. Springer. https://doi.org/10.1007/978-3-540-78800-3_24

[24] Danny Dolev and Andrew Chi-Chih Yao. 1983. On the security of public key protocols. *IEEE Trans. Inf. Theory* 29, 2 (1983). https://doi.org/10.1109/TIT.1983.1056650

[25] Cezara Dragoi, Thomas A. Henzinger, Helmut Veith, Josef Widder, and Damien Zufferey. 2014. A Logic-Based Framework for Verifying Consensus Algorithms. In *VMCAI (LNCS, Vol. 8318)*. Springer. https://doi.org/10.1007/978-3-642-54013-4_10

[26] Cezara Dragoi, Thomas A. Henzinger, and Damien Zufferey. 2016. PSync: a partially synchronous language for fault-tolerant distributed algorithms. In *POPL*. ACM. https://doi.org/10.1145/2837614.2837650

[27] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. 1988. Consensus in the presence of partial synchrony. *J. ACM* 35, 2 (1988). https://doi.org/10.1145/42282.42283

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

[29] Jeremiah Griffin, Mohsen Lesani, Narges Shadab, and Xizhe Yin. 2020. TLC: temporal logic of distributed components. *Proc. ACM Program. Lang.* 4, ICFP (2020). https://doi.org/10.1145/3409005

[30] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *SOSP*. ACM. https://doi.org/10.1145/2815400.2815428

[31] The Iris Project. 2022. Iris: a Higher-Order Concurrent Separation Logic Framework, implemented and verified in the Coq proof assistant. https://iris-project.org/ Online; last accessed 29 April 2024.

[32] Igor Konnov, Marijana Lazic, Ilina Stoilkovska, and Josef Widder. 2023. Survey on Parameterized Verification with Threshold Automata and the Byzantine Model Checker. *Log. Methods Comput. Sci.* 19, 1 (2023). https://doi.org/10.46298/LMCS-19(1:5)2023

[33] Igor V. Konnov, Marijana Lazic, Helmut Veith, and Josef Widder. 2017. A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. In *POPL*. ACM. https://doi.org/10.1145/3009837.3009860

[34] Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. 2020. Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems. In *ESOP (LNCS, Vol. 12075)*. Springer. https://doi.org/10.1007/978-3-030-44914-8_13

[35] Leslie Lamport. 1977. Proving the Correctness of Multiprocess Programs. *IEEE Trans. Software Eng.* 3, 2 (1977). https://doi.org/10.1109/TSE.1977.229904

[36] Leslie Lamport. 2002. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley. http://research.microsoft.com/users/lamport/tla/book.html

[37] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. 1982. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* 4, 3 (1982). https://doi.org/10.1145/357172.357176

[38] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR (LNCS, Vol. 6355)*. Springer. https://doi.org/10.1007/978-3-642-17511-4_20

[39] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. 2016. Chapar: certified causally consistent distributed key-value stores. In *POPL*. ACM. https://doi.org/10.1145/2837614.2837622

[40] Pierre Letouzey. 2008. Extraction in Coq: An Overview. In *CiE (LNCS, Vol. 5028)*. Springer. https://doi.org/10.1007/978-3-540-69407-6_39

[41] Benoît Libert, Marc Joye, and Moti Yung. 2014. Born and raised distributively: fully distributed non-interactive adaptively-secure threshold signatures with short shares. In *PODC*. ACM. https://doi.org/10.1145/2611462.2611498

[42] Giuliano Losa and Mike Dodds. 2020. On the Formal Verification of the Stellar Consensus Protocol. In *2nd Workshop on Formal Methods for Blockchains, FMBC@CAV (OASIcs, Vol. 84)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. https://doi.org/10.4230/OASICS.FMBC.2020.9

[43] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. 2016. A Secure Sharding Protocol For Open Blockchains. In *CCS*. ACM. https://doi.org/10.1145/2976749.2978389

[44] Hammurabi Mendes, Maurice Herlihy, Nitin Vaidya, and Vijay K Garg. 2015. Multidimensional agreement in byzantine systems. *Distributed Computing* 28, 6 (2015). https://doi.org/10.1007/S00446-014-0240-5

[45] Oded Padon, Jochen Hoenicke, Giuliano Losa, Andreas Podelski, Mooly Sagiv, and Sharon Shoham. 2018. Reducing liveness to safety in first-order logic. *Proc. ACM Program. Lang.* 2, POPL (2018). https://doi.org/10.1145/3158114

[46] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: safety verification by interactive generalization. In *PLDI*. ACM. https://doi.org/10.1145/2908080.2908118

[47] Karl Palmskog, Milos Gligoric, Lucas Pena, Brandon Moore, and Grigore Roşu. 2018. Verification of Casper in the Coq proof assistant. (2018). https://core.ac.uk/download/pdf/161954227.pdf.

[48] George Pîrlea. 2019. *Toychain: Formally Verified Blockchain Consensus.* Master's thesis. University College London.

[49] George Pîrlea. 2021. Errors Found in Distributed Protocols. https://github.com/dranov/protocol-bugs-list Online; last accessed 29 April 2024.

[50] George Pîrlea and Ilya Sergey. 2018. Mechanising Blockchain Consensus. In *CPP*. ACM. https://doi.org/10.1145/3167086

[51] Ruzica Piskac, Leonardo Mendonça de Moura, and Nikolaj Bjørner. 2010. Deciding Effectively Propositional Logic Using DPLL and Substitution Sets. *J. Autom. Reason.* 44, 4 (2010). https://doi.org/10.1007/S10817-009-9161-6

[52] Amir Pnueli. 1977. The Temporal Logic of Programs. In *FOCS*. IEEE Computer Society. https://doi.org/10.1109/SFCS.1977.32

[53] Longfei Qiu, Yoonseung Kim, Ji-Yong Shin, Jieung Kim, Wolf Honore, and Zhong Shao. 2024. LiDO: Linearizable Byzantine Distributed Objects with Refinement-Based Liveness Proofs. *PACMPL* 8, PLDI (2024). https://doi.org/10.1145/3656423

[54] Vincent Rahli, Ivana Vukotic, Marcus Völp, and Paulo Jorge Esteves Veríssimo. 2018. Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq. In *ESOP (LNCS, Vol. 10801)*. Springer. https://doi.org/10.1007/978-3-319-89884-1_22

[55] Michael K. Reiter. 1994. Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart. In *CCS*. ACM. https://doi.org/10.1145/191177.191194

[56] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. IEEE. https://doi.org/10.1109/LICS.2002.1029817

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

[58] Upamanyu Sharma, Ralf Jung, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2023. Grove: a Separation-Logic Library for Verifying Distributed Systems. In *SOSP*. ACM. https://doi.org/10.1145/3600006.3613172

[59] Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. 2018. Modularity for decidability of deductive verification with applications to distributed systems. In *PLDI*. ACM. https://doi.org/10.1145/3192366.3192414

[60] The Coq Development Team. 2024. The Coq Reference Manual – Release 8.19.0. https://coq.inria.fr/doc/V8.19.0/refman.

[61] Søren Eller Thomsen and Bas Spitters. 2021. Formalizing Nakamoto-Style Proof of Stake. In *CSF*. IEEE. https://doi.org/10.1109/CSF51468.2021.00042

[62] Ivana Vukotic, Vincent Rahli, and Paulo Jorge Esteves Veríssimo. 2019. Asphalion: trustworthy shielding against Byzantine faults. *Proc. ACM Program. Lang.* 3, OOPSLA (2019). https://doi.org/10.1145/3360564

[63] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *PLDI*. ACM. https://doi.org/10.1145/2737924.2737958

[64] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas E. Anderson. 2016. Planning for change in a formal verification of the Raft consensus protocol. In *CPP*. ACM. https://doi.org/10.1145/2854065.2854081

[65] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *PODC*. ACM. https://doi.org/10.1145/3293611.3331591

[66] Qiyuan Zhao, George Pîrlea, Karolina Grzeszkiewicz, Seth Gilbert, and Ilya Sergey. 2024. *Bythos: Compositional Verification of Composite Byzantine Protocols. Software Artefact.* https://doi.org/10.5281/zenodo.12787570

Received 2024-04-29; accepted 2024-07-04