Model Checking Nondeterministic Behaviours in the Tendermint Byzantine Fault Tolerant Blockchain Consensus Protocol

Yisong Yu[1,2], Zhe Hou[3], Naipeng Dong[4], and Jin Song Dong[2]

[1] Ningbo University, Ningbo City, Zhejiang Province 315211, China
[2] National University of Singapore, 21 Lower Kent Ridge Road 119077, Singapore
[3] Griffith University, 170 Kessels Rd, Nathan QLD 4111, Australia
[4] The University of Queensland, St Lucia QLD 4072, Australia
e0792458@u.nus.edu,n.dong@uq.edu.au,z.hou@griffith.edu.au,dcsdjs@nus.edu.sg

Abstract. We investigated the Tendermint protocol, a core Byzantine Fault Tolerance (BFT) consensus engine for the Cosmos Blockchain. When modelling this protocol, we faced significant challenges in the computational performance of verification. To mitigate the state-space explosion issue, we improve the model with optimisation techniques, such as partial-order reduction and role-based symmetry reduction. Through verification, we discovered vulnerabilities in the design of Tendermint, and we proposed fixes that regained both the safety and liveness properties. This paper describes our modelling techniques and optimisations, analyses the vulnerabilities and fixes, and discusses how the optimisations impact verification runtime and results.

Keywords: Model Checking · Formal Verification · Blockchain · Consensus Protocol · Byzantine Fault Tolerance

1 Introduction

Blockchain is gaining widespread adoption, serving as the backbone of decentralized systems, such as cryptocurrencies, decentralized finance, and supply chain management. It fundamentally shifts how data is stored, verified, and processed to eliminate single points of failure, making it resilient to fraud, tampering, or unauthorized operations. These qualities heavily depend on the underlying consensus mechanism, which plays a critical role in validating data changes and maintaining the security and consistency of the applications built on top. The robustness and efficiency of consensus mechanisms directly impact the reliability, scalability, and overall performance of blockchain-based decentralized systems.

Despite their widespread use, rigorous proof of Blockchain consensus mechanisms is scarce. This is largely due to the complexity of achieving consensus, which requires cooperative execution of geographically distributed nodes exchanging information asynchronously. Challenges arise from issues such as partial connectivity of the network, unreliable communication channels and malicious

nodes (with Byzantine behaviour). The concurrent and nondeterministic nature of node interactions introduces significant challenges for formal verification.

We aim to investigate the capability of formal verification, particularly automatic verification technology, for consensus mechanisms, taking the Tendermint consensus protocol as a case study. We focus on model checking technique, which is widely used in existing consensus verification efforts. Tendermint was chosen due to its practical significance—it powers numerous real-world blockchain platforms like Cosmos. Also, unlike Bitcoin's probabilistic protocols, Tendermint's non-probabilistic nature makes it more feasible for automatic proof.

Previous research on verifying Tendermint consensus protocol has primarily focused on verifying safety (maintaining data integrity) because liveness (ensuring continuing operation) verification poses significant challenges due to the extensive state space when naively translating the pseudocode into formal specifications. Our goal is to verify both safety and liveness in the presence of Byzantine nodes. To overcome the challenges, we apply state space reduction techniques such as partial order reduction and role-based symmetry reduction, which are crucial for efficient verification of both safety and liveness properties.

We have successfully verified the correctness of this algorithm using the PAT model checker [23,8] in the CSP# formal language, and we believe that we are the first to have verified both the safety and liveness properties of Tendermint. The verification is non-trivial due to the state-space explosion challenges, and the optimisation techniques in this work have the potential to be adopted in verifying other consensus protocols. As a highlight, during our investigation, we identified a critical vulnerability hidden within the Tendermint design that could be exploited to orchestrate a denial-of-service attack, potentially jeopardizing the entire system. This finding emphasizes the importance of rigorous validation of blockchain security. Moreover, we discussed the generalization of our reduction techniques beyond this case study.

In summary, this work makes the following contributions:

- We modelled the Tendermint protocol and proposed optimisations to make verification efficient. We proved that our optimisations preserve the correct semantics of the model, and they are also applicable to similar BFT protocols.
- We identified a vulnerability of Tendermint, and we give an attack demonstration as well as a fix.
- With the help of model optimisations, we verified the Tendermint protocol's safety and liveness after the vulnerability was patched.

2 The Tendermint Consensus Protocol

The Tendermint consensus protocol [10] is implemented in the Tendermint Core of the Cosmos SDK, and is a *state machine replication* algorithm [20] that is inspired by the Practical Byzantine Fault Tolerance (PBFT) algorithm [11] and the DLS algorithm [18].

Protocol Overview. The protocol proceeds in rounds, and the communication pattern of each round follows a simple state machine as depicted in Figure 1, re-

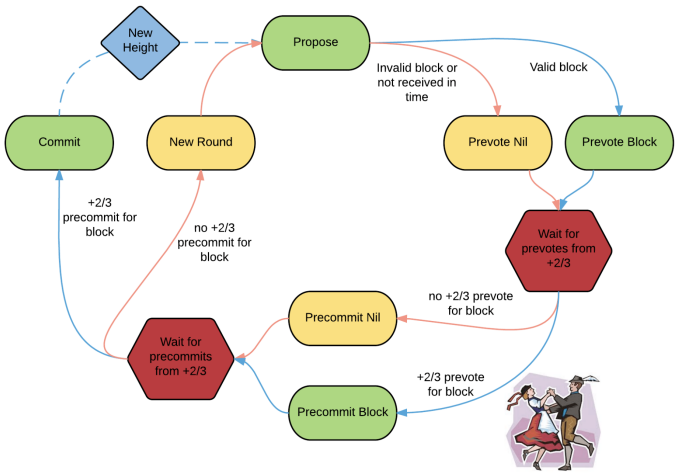

Fig. 1: Overview of Tendermint consensus protocol [24].

sembling the normal case in PBFT [11]. Each round features a proposer selected through a weighted round-robin function based on voting power. Participants are usually referred to as validators which verify the proposed blocks of transactions to be sequentially committed in a chain, each at a new height. Although the state machine diagram seems simple with only ten states, it reflects a limited, single-node perspective, omitting concurrent actions by other nodes. The entire network operation is much more complex, making manual analysis challenging and error-prone.

Protocol Specification. The protocol is detailed in the Tendermint consensus algorithm from the literature [10], shown in Algorithm 1. It uses atomically executed "upon rules", triggered when a local message log accumulates enough messages to satisfy conditions based on validators' voting power. Messages exchanged among nodes include PROPOSAL, PREVOTE, and PRECOMMIT to trigger an upon rules. In addition, the protocol ensures timeliness and reliability through timeout mechanisms (e.g., $timeoutPropose$, $timeoutPrevote$, $timeoutPrecommit$) which are incrementally adjusted for each new round to prevent indefinite delays.

Processes exchange three types of messages to agree on a proposed block of transactions: the proposer sends a PROPOSAL, and nodes respond with PREVOTE and PRECOMMIT messages. PREVOTE and PRECOMMIT carry a small identifier (e.g., a hash of the proposed block) to confirm agreement. To reach a decision, nodes require a PROPOSAL and enough PRECOMMIT messages (i.e., from validators representing at least two-thirds of the voting power). Processes maintain internal variables like $lockedValue$, $lockedRound$, $validValue$, and $validRound$ to track their decision progress and ensure termination and agreement. A correct process updates $validValue$ and $validRound$ when it receives a valid proposal backed by sufficient votes. This guarantees that if a process locks on a value, all correct processes will eventually converge on that decision.

Algorithm 1: The Tendermint Algorithm [10]

1: **Initialization:**
2: $h_p, round_p := 0$; $step_p \in \{propose,\ prevote,\ precommit\}$;
3: $validRound_p := -1$;
4: $decision_p[], lockedValue_p, lockedRound_p, validValue_p := nil$;
5: **upon** start **do** *StartRound(0)*
6: **Function** *StartRound(round)*:
7: $round_p \leftarrow round$; $step_p \leftarrow step$;
8: **if** proposer$(h_p,\ round_p) = p$ **then**
9: **if** $validValue_p \neq nil$ **then**
10: $proposal \leftarrow validValue_p$;
11: **else**
12: $proposal \leftarrow getValue()$;
13: **broadcast** $\langle$PROPOSAL, $h_p,\ round_p,\ proposal,\ validRound_p\rangle$;
14: **else**
15: **schedule** *OnTimeoutPropose*$(h_p,\ round_p)$ to be executed **after** $timeoutPropose(round_p)$;
16: **upon** $\langle$PROPOSAL, $h_p,\ round_p,\ v,\ -1\rangle$ **from** proposer$(h_p,\ round_p)$ **while** $step_p = propose$ **do**
17: **if** $valid(v) \wedge (lockedRound_p = -1 \vee lockedValue_p = v)$ **then**
18: **broadcast** $\langle$PREVOTE, $h_p,\ round_p,\ id(v)\rangle$;
19: **else**
20: **broadcast** $\langle$PREVOTE, $h_p,\ round_p,\ nil\rangle$;
21: $step_p \leftarrow prevote$;
22: **upon** $\langle$PROPOSAL, $h_p,\ round_p,\ v,\ vr\rangle$ **from** proposer$(h_p,\ round_p)$ **AND** $2f + 1\ \langle$PREVOTE$, h_p, vr, id(v)\rangle$ **while** $step_p = propose \wedge (vr \geq 0 \wedge vr < round_p)$ **do**
23: **if** $valid(v) \wedge (lockedRound_p \leq vr \vee lockedValue_p = v)$ **then**
24: **broadcast** $\langle$PREVOTE, $h_p,\ round_p,\ id(v)\rangle$;
25: **else**
26: **broadcast** $\langle$PREVOTE, $h_p,\ round_p,\ nil\rangle$;
27: $step_p \leftarrow prevote$;
28: **upon** $2f + 1\ \langle$PREVOTE$, h_p, round_p, *\rangle$ **while** $step_p = prevote$ first time **do**
29: **schedule** *OnTimeoutPrevote*$(h_p,\ round_p)$ to be executed **after** $timeoutPrevote(round_p)$;

The gossip communication property ensures messages are propagated across all nodes, enabling processes to synchronize their decisions. The termination mechanism ensures that, even in prolonged rounds, the protocol will eventually finalize a decision. This is achieved without extra messages, making Tendermint efficient and reliable for consensus in distributed systems.

Correctness Criteria. We adopt the correctness criteria from distributed systems [11] in the Tendermint setting as follows:

Safety requires that correct validators all decide on the same value, or formally $\forall p, q \in validators : honest(p) \wedge honest(q) \Rightarrow decision[h_p] = decision[h_q]$.

Liveness requires that all correct validators must eventually decide on a value, or formally $\forall p \in validators : \exists v \in values : honest(p) \Rightarrow decision[h_p] = v$.

3 Our Modelling of Tendermint

Modelling and verifying the Tendermint protocol presented tremendous challenges, especially in verification time. Faithfully modelling the protocol leads to

Algorithm 1: The Tendermint Algorithm (Continued) [10]

```
32: upon ⟨PROPOSAL, h_p, round_p, v, ∗⟩ from proposer(h_p, round_p) AND
    2f + 1 ⟨PREVOTE, h_p, round_p, id(v)⟩ while valid(v) ∧ step_p ≥ prevote for the first
    time do
33:     if step_p = prevote then
34:         lockedValue_p ← v; lockedRound_p ← round_p;
35:         broadcast ⟨PRECOMMIT, h_p, round_p, id(v)⟩;
36:         step_p ← precommit;
37:     validValue_p ← v; validRound_p ← round_p;

38: upon 2f + 1 ⟨PREVOTE, h_p, round_p, nil⟩ while step_p = prevote do
39:     broadcast ⟨PRECOMMIT, h_p, round_p, nil⟩;
40:     step_p ← precommit;

41: upon 2f + 1 ⟨PRECOMMIT, h_p, round_p, ∗⟩ for the first time do
42:     schedule OnTimeoutPrecommit(h_p, round_p) to be executed after
          timeoutPrecommit(round_p);

43: upon ⟨PROPOSAL, h_p, r, v, ∗⟩ from proposer(h_p, r) AND
    2f + 1 ⟨PRECOMMIT, h_p, r, id(v)⟩ while decision_p[h_p] = nil do
44:     if valid(v) then
45:         decision_p[h_p] = v; h_p ← h_p + 1;
46:         reset lockedRound_p, lockedValue_p, validRound_p, validValue_p to initial values
              and empty message log;
47:         StartRound(0);

48: upon f + 1 ⟨∗, h_p, round, ∗, ∗⟩ with round > round_p do
49:     StartRound(0);

50: Function OnTimeoutPropose(height, round):
51:     if height = h_p ∧ round = round_p ∧ step_p = propose then
52:         broadcast ⟨PREVOTE, h_p, round_p, nil⟩;
53:         step_p ← prevote;

54: Function OnTimeoutPrevote(height, round):
55:     if height = h_p ∧ round = round_p ∧ step_p = prevote then
56:         broadcast ⟨PRECOMMIT, h_p, round_p, nil⟩;
57:         step_p ← precommit;

58: Function OnTimeoutPrecommit(height, round):
59:     if height = h_p ∧ round = round_p then StartRound(round_p + 1);
```

non-termination within a few days. Realising that modelling is an art, we went through 13 versions of modelling; each one included some optimisations to improve runtime to finally achieve a realistic model that is efficient in verification. This section begins with an overview of the overall modelling approach, followed by a detailed analysis of a vulnerability identified during the model checking of our initial, unoptimized model. Finally, we present a walkthrough of the key optimizations that brought the most significant improvements to our verification process, addressing the inherent state-space explosion issue that persists even in the patched model.

3.1 Protocol Assumptions

Processor: The protocol considers a system of spatially separated ***asynchronous*** processors that communicate with each other by exchanging messages because they do not belong to the same administrative domain and are therefore not directly connected. Each process has some voting power, usually proportional

to the amount of stakes (e.g., cryptocurrencies) it commits to participate in the consensus as a validator.

Network: The protocol assumes a *reliably authenticated* ***partially synchronous*** network on which processors communicate with each other. Here, "reliable" implies that messages cannot be lost, duplicated, altered, or corrupted during their transmission through the communication channels, although they may be delivered out of order, "authenticated" means the recipient can verify the identity of the sender, and "partially synchronous" means messages will be eventually delivered within an upper bound of delay.

Communication: The protocol assumes the following auxiliary *communication property* that captures *gossip-based* nature of communication, where GST is Global Stabilization Time and Δ is the upper bound of delay: If a correct node sends some message m at time t, all correct nodes will receive m before $max\{t, GST\} + \Delta$. If a correct node receives some message m at time t, all correct nodes will receive m before $max\{t, GST\} + \Delta$.

Failure Model: Two types of participants are considered in the design of a consensus algorithm [9]: *honest/non-faulty* nodes always engage in benign behaviours by strictly following the pre-defined protocol, and *corrupted/faulty* nodes may randomly crash, or have malicious (Byzantine) intent, e.g., withholding messages, sending invalid messages, and violating expected timing assumptions.

Computation Model: The protocol assumes an atomic execution of each logical entity in the form of an *upon rule* is truly atomic. In addition, a simple ***send*** and its aggregated counterpart ***broadcast*** both operate similarly atomically.

3.2 Introduction to CSP# and PAT

The input language of PAT is CSP#, whose syntax is shown in Definition 1. We chose PAT due to its support for state variables, common programming constructs, and moreover external libraries and functions.

Definition 1 (Syntax of Processes in CSP#).

$$P,\ Q\ ::= Stop \mid Skip \mid e \rightarrow P \mid e\{prog\} \rightarrow P \mid c!exp \rightarrow P \mid c?x \rightarrow P \mid P[]Q \mid if(b)\{P\}\ else\ \{Q\} \mid ifa(b)\{P\}\ else\ \{Q\} \mid P;Q \mid P||Q \mid P|||Q$$

Stop and *Skip* are processes denoting inaction and termination, respectively. Process $e \rightarrow P$ engages in an atomic event e first and then behaves as process P. The event is allowed to attach an atomically executed program, denoted as $e\{prog\} \rightarrow P$. Channel communication is supported: $c!exp \rightarrow P$ denotes sending exp over channel c, while $c?x \rightarrow P$ denotes reading from channel c and referring to the message as x. Two types of choices are supported: $P[]Q$ denotes unconditional choice, and $if(b)\{P\}\ else\ \{Q\}$ is conditional branching, where b is

```
StartRound(p, round) =
 reset_all_flags_in_effective_rounds_for_process.p{effective_rounds[p]=0;} ->
 update_round_and_step_for_process_in_round.p.round{rounds[p]=round; steps[p]=
     PROPOSE;} ->
 ifa (round%N==p) {
  ifa (valid_values[p]!=NIL) {
   update_proposal_for_process_in_round.p.round{proposals[p]=valid_values[p];} ->
     Skip}
   else {
    update_proposal_for_process_in_round.p.round{
     if (honest_processes[p]==true) {proposals[p]=DECISION_T;}
     else {proposals[p]=DECISION_F;}} -> Skip};
   BroadcastProposalMessage(p, rounds[p], proposals[p], valid_rounds[p])
  } else {ScheduleOnTimeoutPropose(p, rounds[p])};
```

Fig. 2: Process StartRound.

```
BroadcastProposalMessage(p, round, proposal, valid_round) =
 broadcast_proposal_message_to_all_processes_from_process.p{message_log.AddProposal(
     new ProposalMessage(p, round, proposal, valid_round));} -> Skip;
```

Fig. 3: Process BroadcastProposalMessage.

a boolean expression. $ifa(b)\{P\}$ *else* $\{Q\}$ is a variation of conditional branching that performs the condition checking and first operation of P/Q together. There are three types of process relations: Process $P;Q$ behaves as P until P terminates and then behaves as Q. The parallel composition of two processes is written as $P||Q$, where P and Q may communicate via multi-party event synchronisation. If P and Q only communicate through variables, then it is written as $P|||Q$.

3.3 Overall Modelling

The process *StartRound* in Figure 2 models the function **StartRound** from line 7 to line 17 of the protocol. It resets $effective\_rounds[p]$, updates $rounds[p]$ and $steps[p]$, and either updates $proposals[p]$ (to $valid\_values[p]$, $DECISION\_T$, or $DECISION\_F$) or schedules $OnTimeoutPropose$, depending on whether p is the proposer. It prevents reverting to a previous round if $round < rounds[p]$.

The process $BroadcastProposalMessage$, shown in Figure 3, models the behaviour of broadcasting the $PROPOSAL$ message by a proposer (line 15 of the protocol). It wraps the proposal data into a $ProposalMessage$ instance from the C# library, then adds it to the message log by calling the *AddProposal* method on the $message\_log$ variable, completing the broadcast. The other messages, such as $PREVOTE$ and $PRECOMMIT$ are modelled similarly.

Then each upon rule from line 18 to line 52 of the protocol is modelled following their corresponding algorithms. To illustrate, we take one upon rule (line 32-39 of the protocol) as an example, and show our non-trivial modelling techniques in Figure 8 (detailed in Section 3.5). We omit the details of other upon rules due to limited space (for details, see the online code).

The process $ScheduleOnTimeoutPropose$, shown in Figure 4, models how a process p in round $round$ schedules the handler $OnTimeoutPropose$ (line 53-56 of the protocol). This allows p to simulate both the expiration and non-expiration of the timer, controlled by conditional expressions composed with a choice operator.

If the timeout for the $round$ ($INIT\_TIMEOUT\_PROPOSE + round * TIMEOUT\_DELTA$) meets or exceeds a predefined bound $BOUND\_DELTA$

```
ScheduleOnTimeoutPropose(p, round) =
[BOUND_DELTA>INIT_TIMEOUT_PROPOSE+round*TIMEOUT_DELTA
|| honest_processes[round%N]!=honest_processes[p]]OnTimeoutPropose(p, round) []
[honest_processes[round%N]==honest_processes[p]]Skip;
OnTimeoutPropose(p, round) =
 ifa (round==rounds[p]&&steps[p]==PROPOSE) {
  BroadcastPrevoteMessage(p, round, NIL);
  update_step_for_process_in_round.p.round{steps[p]=PREVOTE;} -> Skip};
```

Fig. 4: Process (Schedule)OnTimeoutPropose.

for message delivery in a partially synchronous network, and if p's honesty matches the proposer's in that round, p cannot choose to timeout. In this case, the timer will not expire before p receives a valid $PROPOSAL$ message. This prevents p from prematurely sending a $PREVOTE$ message for nil before the proper rule is invoked.

However, if these conditions are not met, p may opt to timeout, even if the $PROPOSAL$ message is cached in the message log. This model's possible behaviours simulate scenarios where p pretends not to have received the message on time, thus covering all potential timeout behaviours.

The other two functions $onTimeoutPrevote$ (line 57-60 of the protocol) and $onTimeoutPrecommit$ (line 61-62) are modelled in the same manner.

The entire process of Tendermint is shown in Figure 7, where the left side shows a straightforward modelling that leads to state exploration and right hand side shows our final modelling with partial-order reduction.

3.4 Identified Vulnerability and Fix

This vulnerability was revealed when model checking our early model without any optimization strategies. A counterexample trace was generated, showing that the execution of $StartRound(p)$ may be delayed after a given process p has advanced into a later round.

This vulnerability could occur if a node is reverted from a later round ($round \neq 0$) back to $round = 0$ by delaying the initial $StartRound(0)$ execution. This flaw could deadlock the entire system as the protocol does not explicitly require the function to be completed before other rules are executed, making it vulnerable.

The space-time diagram in Figure 5 illustrates how a denial-of-service (DoS) attack could exploit the vulnerability. The horizontal axis represents time, segmented into rounds and steps, while the vertical axis represents processes. Events (message send/receive) are marked with dots, and coloured arrows depict message transmission between processes. Each color indicates a specific message type: orange for $PROPOSAL$ messages, brown for $PREVOTE$ messages for $id(v)$, yellow for $PREVOTE$ messages for nil, purple for $PRECOMMIT$ messages for $id(v)$, and blue for $PRECOMMIT$ messages for nil. The grey text above each event shows which function or rule is activated, leading to the corresponding event below. In this diagram, synchronized message arrivals are shown, although in reality, communication delays are often unstable.

The DoS attack occurs by exploiting the system's vulnerability, where a delayed $StartRound(0)$ invocation can revert nodes back to $round = 0$, disrupting the consensus and eventually deadlocking the system. This example highlights

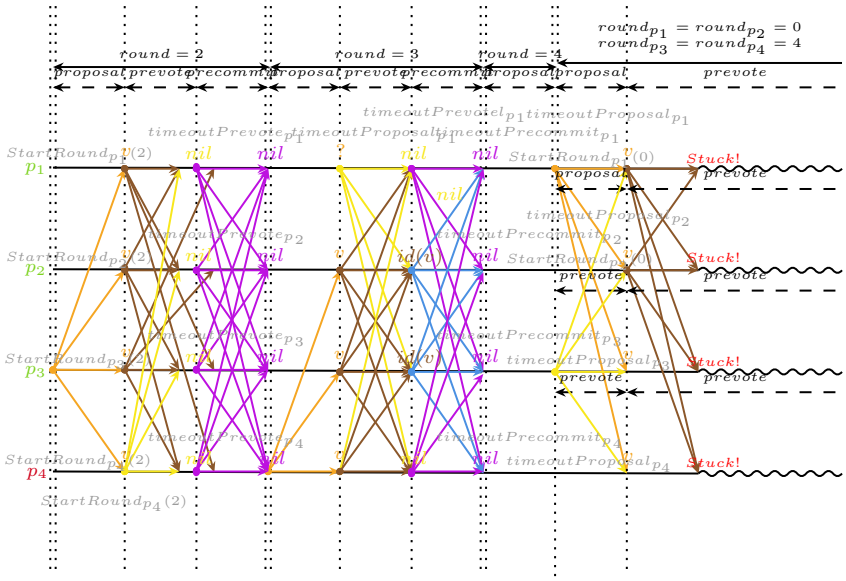

Fig. 5: An attack example demonstrating the identified vulnerability.

how the absence of a strict requirement for completing certain protocol functions before others enable the attack, making it a significant vulnerability.

In this scenario, all nodes in green are honest, except for p_4 in red, which is Byzantine and behaves arbitrarily. p_4 can delay functions or message transmissions, selectively withhold messages, or send conflicting ones, disrupting the consensus process without completely halting the system. More specifically:

1. Four processes have completed two rounds of communication without reaching consensus and are starting round 2, with $StartRound(0)$ for p_1 and p_2 delayed by the adversary since round 0.
2. p_3 and p_4 invoke $StartRound(2)$, and p_3 broadcasts a $PROPOSAL$ message for v. All validators schedule $OnTimeoutPropose(0, 2)$.
3. After receiving the $PROPOSAL$ message for v, all processes send $PREVOTE$ messages for $id(v)$, except p_4, who votes for nil, creating a split vote.
4. The timers $timeoutPrevote(2)$ expire before enough $PREVOTE$ messages are received, and all processes send $PRECOMMIT$ messages for nil.
5. After receiving sufficient $PRECOMMIT$ messages for nil, all processes advance to round 3, with p_4 as proposer, broadcasting $PROPOSAL(v)$ to all except p_1.
6. p_2, p_3, and p_4 send $PREVOTE(id(v))$, while p_4 sends $PREVOTE(nil)$ to p_1, who then votes nil upon timeout.
7. p_2 and p_3 send $PRECOMMIT(id(v))$, while p_1 and p_4 send $PRECOMMIT(nil)$, resulting in another split vote.
8. Upon $timeoutPrecommit(3)$ expiry, all processes start round 4, with p_4 withholding $PROPOSAL$ messages.
9. The adversary relinquishes control over p_1 and p_2, reverting them to round 0, causing p_1 to broadcast a $PROPOSAL$ for v.

```
StartRoundFixed(p, round) = ifa (round>=rounds[p]) {StartRound(p, round)};
```

Fig. 6: Process StartRound (Fixed ver.).

10. p_1 and p_2 send obsolete $PREVOTE(id(v))$ messages for round 0, which are discarded by others.
11. The system reaches deadlock with p_1 and p_2 stuck in round 0, and p_3 and p_4 in round 4, resulting in a denial-of-service attack.

To fix this vulnerability and prevent a malicious entity from exploiting it, we need to wrap the entire body of the function *StartRound* into an if statement that checks if the round *round* to be entered is not smaller than the round the process p is currently in, i.e., if $round >= rounds[p]$, as shown in Figure 6.

3.5 Reduction Techniques

With the vulnerability identified above addressed, it remains crucial to verify the patched protocol to ensure that the fix does not introduce new bugs and that the protocol continues to satisfy both safety and liveness properties. However, model-checking the corrected version still presents a significant challenge due to the state-space explosion problem, as the number of states grows combinatorially with all possible interleavings of rules across processes, often causing the verification engine to exceed memory limits and abort.

Among those well-known strategies for state space minimization, we applied two key optimizations that are considered most effective in our model, partial-order reduction and symmetry reduction, designed to address the intractability of model checking highly concurrent distributed protocols like Tendermint; we ensure that the optimizations reduce state space without omitting any potential counterexamples from the full model in tools like PAT.

Partial-order Reduction (POR) POR reduces state space by representing multiple interleaving executions with a single partial order execution, leveraging the connection between interleaving semantics, which treats executions differing only in independent transition order as equivalent, and partial order semantics, which orders events causally. This mitigates the *state space explosion* problem in concurrent systems by generating a compact state space that ensures counterexamples, if present, are preserved in the reduced state space.

We present an example of applying POR in Figure 7, where our model's separation into two distinct processes, $TendermintBootstrap$ and $Tendermint$, shown on the bottom, is preferred over the more intuitive but naive approach of interleaving all participating validators into a single process, shown on the top. In our optimized model, each validator is represented by its own process, $Validator$, consisting of a general choice of those processes denoting upon rules, which is then general-choiced with processes denoting other validators in $Tendermint$, which is then sequentially composed with the process *StartRound* of all validators to enforce a total ordering on the execution sequence of the function *StartRound* at the beginning of model checking, given that this approach significantly reduces the state space—by an order of magnitude—by mitigating the

```
Tendermint() = ||| p: {0..N-1} @ (
  StartRound(p,0) ||| DaemonThread(p) ||| Mutex(p) |||
  UponProposalNewValue(p) ||| UponProposalOldValue(p) |||
  UponSufficientPrevoteAny(p, 0) ||| UponSufficientPrevoteValue(p, 0) |||
  UponSufficientPrevoteNil(p) ||| UponSufficientPrecommitAny(p, 0) |||
  UponSufficientPrecommitValue(p) ||| UponSufficientMessageAny(p, 0));
```

```
Validator(p) = UponProposalValue(p) []
  UponSufficientPrevoteAny(p) [] UponSufficientPrevoteValue(p) []
  UponSufficientPrevoteNil(p) [] UponSufficientPrecommitAny(p) []
  UponSufficientPrecommitValue(p) [] UponSufficientMessageAny(p);
TendermintBootstrap() = StartRound(0, 0); StartRound(1, 0);
  StartRound(2, 0); StartRound(3, 0); Tendermint();
Tendermint() = Validator(0) [] Validator(1) [] Validator(2) [] Validator(3) []
               OnTimeoutPrevoteManual() [] OnTimeoutPrecommitManual();
```

Fig. 7: A comparison of example code without (top) & with (bottom) POR.

complexity introduced by the interleaving operator, which otherwise generates all possible execution orders, thereby successfully addressing the timeout and out-of-memory issues encountered in earlier models that naively modelled *concurrency*. The decision to impose a predefined order in $TendermintBootstrap$ and replace interleaving with general choice operators in $Tendermint$ is driven by the observation that nearly all transitions (*i.e.*, operations at each step) are *local/internal*. Excluding pure events that do not introduce side effects to the global states, these transitions only mutate local variables, neither interfering with other processes nor disabling each other (*i.e.*, satisfying *enabledness*), and can be commutated with each other while leading to the same final state (*i.e.*, satisfying *commutativity*), indicating their *independence*.

Correctness of our POR. Next, we discuss the correctness of our partial-order reduction. We adopt the definitions from Clarke et al.'s partial-order reduction technique [13,14]. This technique operates within a model checking algorithm applied to a transition system modelled as a Kripke structure $\mathcal{K} = \langle S, I, R, L, AP \rangle$ over a set of atomic propositions AP where S is a set of states, $I \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is a transition relation between states, and $L : S \to 2^{AP}$ is a labelling function that maps each state to a subset of atomic propositions. A transition α is considered *enabled* in state s if there is another state s' where $(s, s') \in R$. If only one transition is enabled from s, α is a *deterministic transition*, expressible as $s' = \alpha(s)$. Additionally, execution paths are assumed infinite, even if deadlocks occur, by introducing self-transitions. This framework supports reasoning about transition *reordering* in model checking algorithms.

Definition 2 (Independence Relation). *The* independence relation $IR \subseteq R \times R$ *is a symmetric and anti-reflexive relation on transitions. A pair of* independent *transitions* $(\alpha, \beta) \in IR$ *satisfies the following for each state* $s \in S$*:*

- Enabledness*: If* $\alpha, \beta \in enabled(s)$*, then* α *and* β *remain enabled in each other's resulting state.*
- Commutativity*: If* $\alpha, \beta \in enabled(s)$*, then executing* α *followed by* β *gives the same result as executing* β *followed by* α*.*

We introduce the concept of *invisible* transitions to address the variability in verification outcomes caused by reordered execution sequences with independent

transitions. These transitions help clarify how specifications differentiate between states, despite the states appearing equivalent based on their transition orders.

Definition 3 (Invisible Transitions). *A transition $\alpha \in R$ is termed* invisible *concerning a subset AP' of atomic propositions AP if its execution between any two states does not alter their labellings. This means for states s and s' where $s' = \alpha(s)$, it holds that $L(s) \cap AP' = L(s') \cap AP'$.*

Definition 4 (Stuttering Equivalence of Paths). *Two infinite paths $\sigma = s_0, s_1, s_2, ...$ and $\rho = r_0, r_1, r_2, ...$ are* stuttering equivalent*, denoted $\sigma \equiv_{st} \rho$, if there exist sequences of indices $0 = i_0 < i_1 < ...$ and $0 = j_0 < j_1 < ...$ such that for all $k \geq 0$, $L(s_{i_k}) = L(s_{i_{k+1}}) = L(r_{j_k}) = L(r_{j_{k+1}})$, indicating matching labelled subsequences in both paths.*

Definition 5 (Stuttering Invariance of LTL Formulas). *An LTL$_{-X}$ formula ψ (without "next") is* invariant under stuttering *if for any two* stuttering equivalent *paths σ and σ' (i.e., $\sigma \equiv_{st} \sigma'$), it holds that $\sigma \models \psi$ iff $\sigma' \models \psi$.*

Definition 6 (Stuttering Equivalence of State Transition Systems). *Two state transition systems M and M' are considered* stuttering equivalent *if:*
- *For every path σ starting from an initial state of M, there exists a path σ' from an initial state of M' such that $\sigma \equiv_{st} \sigma'$.*
- *For every path σ' starting from an initial state of M', there exists a path σ from an initial state of M such that $\sigma' \equiv_{st} \sigma$.*

Based on the definitions of *stuttering equivalence* of state transition systems in Definition 6 and *stuttering invariance* of LTL$_{-X}$ formulas in Definition 5, we conclude that if two state transition systems M and M' are *stuttering equivalent*, then $M \models \psi$ iff $M' \models \psi$ for any LTL$_{-X}$ formula ψ. This justifies the validity of using the partial order reduction technique to create a structure that is *stuttering equivalent* to the original system.

Lemma 1 (Correctness of Partial-Order Reduction). *Given a transition system represented as a Kripke structure $\mathcal{K} = \langle S, I, R, L, AP \rangle$ and the reduced structure $\mathcal{K}_p = \langle S_p, I_p, R_p, L_p, AP \rangle$ via the above partial-order reduction, $\mathcal{K}$ and $\mathcal{K}_P$ are stuttering equivalent.*

Theorem 1 (Correctness of POR Tendermint Model). *Given the Tendermint consensus protocol modelled as $\mathcal{K} = \langle S, I, R, L, AP \rangle$ and the reduced model $\mathcal{K}_p = \langle S_p, I_p, R_p, L_p, AP \rangle$ via partial-order reduction, if a counterexample exists in the state space of $\mathcal{K}$ then it also exists in the state space of $\mathcal{K}_p$.*

Role-based Symmetry Reduction (SR) Structural symmetries, such as replicated components in concurrent systems, often go unnoticed but can help reduce the *state space explosion* problem. Symmetry reduction treats states that differ only in the arrangement of identical processes as equivalent. This technique collapses these symmetric states, called *orbits*, into a single representative during

```
UponSufficientPrevoteValue(p) =
[decisions[p]==NIL && steps[p]>=PREVOTE && (effective_rounds[p]/2)%2==0 &&
     message_log.ContainsProposalAndSufficientPrevotesForPrecommitting(all_messages,
      rounds[p]) &&
 ((honest_processes[p]==true && message_log.GetProposalValue(all_messages, rounds[p]
     )==DECISION_T) ||
 (honest_processes[p]==false && message_log.GetProposalValue(all_messages, rounds[p]
     )==DECISION_F))]
enable_second_flag_in_effective_round_for_process.p{effective_rounds[p]=
     effective_rounds[p]+2;} -> UponSufficientPrevoteValueAuxiliary(p, rounds[p]);
UponSufficientPrevoteValueAuxiliary(p, round) =
ifa (steps[p]==PREVOTE) {
 update_locked_value_and_locked_round_for_process_in_round.p.round{locked_values[p]=
     message_log.GetProposalValue(all_messages, round); locked_rounds[p]=round;} ->
 BroadcastPrecommitMessage(p, round, message_log.GetProposalValue(all_messages,
     round));
 update_step_for_process_in_round.p.round{steps[p]=PRECOMMIT;} -> Skip};
 update_valid_value_round_for.p.round{valid_values[p]=message_log.GetProposalValue(
     all_messages, round); valid_rounds[p]=round;} -> Tendermint();
```

```
UponSufficientPrevoteValue(p) =
[(p==0 || (rounds[p-1]==rounds[p] && steps[p-1]>=PRECOMMIT) || rounds[p-1]>rounds[p]
     ) && decisions[p]==NIL &&
 (effective_rounds[p]/2)%2==0 && steps[p]>=PREVOTE && message_log.
     ContainsProposalAndSufficientPrevotesForPrecommitting(all_messages, rounds[p])
     &&
 ((honest_processes[p]==true && message_log.GetProposalValue(all_messages, rounds[p]
     )==DECISION_T) ||
  (honest_processes[p]==false && message_log.GetProposalValue(all_messages, rounds[p
     ])==DECISION_F))]
enable_second_flag_in_effective_round_for_process.p{effective_rounds[p]=
     effective_rounds[p]+2;} ->
ifa (steps[p]==PREVOTE) {
 update_locked_value_and_locked_round_for_process_in_round.p.rounds[p]{locked_values
     [p]=message_log.GetProposalValue(all_messages, rounds[p]); locked_rounds[p]=
     rounds[p];} ->
 BroadcastPrecommitMessage(p, rounds[p], message_log.GetProposalValue(all_messages,
     rounds[p]));
 update_step_for_process_in_round.p.rounds[p]{steps[p]=PRECOMMIT;} -> Skip};
 update_valid_value_and_valid_round_for_process_in_round.p.rounds[p]{valid_values[p]
     =message_log.GetProposalValue(all_messages, rounds[p]); valid_rounds[p]=rounds[
     p];} -> Tendermint();
```

Fig. 8: A comparison of example code without (top) & with (bottom) SR.

model checking. Unlike partial order reduction, which minimizes independent transition interleavings, symmetry reduction focuses on collapsing equivalent states. It ensures that any counterexample in the full state space is still found in the reduced space, improving efficiency without sacrificing verification accuracy.

We show an example code snippet modified for symmetry reduction in Figure 8 where in addition to the original conditions for the upon rule two extra constraints added to the guarded expression of our corresponding process $UponSufficientPrevoteValue(p)$ with respect to two orthogonal dimensions in data values, namely *round* and *step* serve as the key to a significant reduction of the state space by an order of magnitude, via imposing a strict total ordering on the executions of upon rules by different processes at different steps within the same round (*i.e.*, $p == 0||(rounds[p-1] == rounds[p]\&\&steps[p-1] >= PRECOMMIT)$), as well as those executions of this upon rule similarly by different processes at potentially same or different steps across different rounds (*i.e.*, $p == 0||rounds[p-1] > rounds[p]$), so that unnecessary exploration of symmetric executions may be avoided to enable a faster termination of the model

checking process. Note that other upon rules (omitted here for brevity) also apply the same modifications to their guarded conditions to enforce a total ordering on their executions as well.

Correctness of our SR. We now show, on a high level, the correctness of our symmetry reduction. We again use a Kripke structure $\mathcal{K} = \langle S, I, R, L, AP \rangle$, and its associated behaviours (enabled transitions, deterministic paths, and infinite execution paths). We adopt definitions from prior work [12,15,19] to reason about the correctness of symmetry reduction applied at the model level for distributed protocols modelled into a state transition system.

To formalize the definitions of *symmetric* states and paths, which help in selecting one representative from each equivalence class for analysis, we begin with the concept of a *permutation*. From this, we define the *symmetry group* and *orbit* within a transition system.

Definition 7 (Permutation). *A permutation $\sigma : S \to S$ on a set S is a* bijection *of S onto itself, meaning $(\forall s \in S : \exists_{=1} \sigma(s) \in S) \wedge (\forall \sigma(s) \in S : \exists_{=1} s \in S)$, where $\exists_{=1}$ means exists only one.*

Definition 8 (Symmetry Group). *In a Kripke structure $\mathcal{K} = \langle S, I, R, L, AP \rangle$, a subgroup G of all permutations of S (denoted $Perm(S)$) is a* symmetry group *if every permutation $\sigma \in G$ preserves the transition relation R, meaning $(s_1, s_2) \in R$ iff $(\sigma(s_1), \sigma(s_2)) \in R$. Each $\sigma \in G$ is called an* automorphism *of $\mathcal{K}$.*

Definition 9 (Equivalence Relation & Orbit). *A symmetry group G acting on a Kripke structure $\mathcal{K}$ defines an* equivalence relation $\equiv$ *on S, where $s_1 \equiv s_2$ if $\exists \sigma \in G$ such that $s_2 = \sigma(s_1)$. The* equivalence class $[s] = \{t \mid \exists \sigma \in G : \sigma(s) = t\}$ *is called the* orbit *of s under G, with $rep([s])$ as the representative.*

Definition 10 (Quotient Structure). *Given a Kripke structure $\mathcal{K} = \langle S, I, R, L, AP \rangle$ and a symmetry group G acting on $\mathcal{K}$, the quotient structure for $\mathcal{K}$ modulo G is a Kripke structure $\mathcal{K}_G = \langle S_G, I_G, R_G, L_G, AP \rangle$, where $S_G = \{[s] \mid s \in S\}$, $I_G = \{[s_0] \mid s_0 \in I\}$, $R_G = \{([s], [t]) \mid (s, t) \in R\}$, $L_G([s]) = L(rep([s])) = L(s)$ for all $s \in S$.*

Definition 11 (Invariance Group). *In a Kripke structure $\mathcal{K} = \langle S, I, R, L, AP \rangle$, a symmetry group G is an* invariance group *for an atomic proposition p if: $\forall \sigma \in G : \forall s \in S : L(s) = L(\sigma(s))$*

Definition 12 (Bisimulation Relation). *Given two Kripke structures $\mathcal{K}_1 = \langle S_1, I_1, R_1, L_1, AP \rangle$ and $\mathcal{K}_2 = \langle S_2, I_2, R_2, L_2, AP \rangle$, a binary relation $\mathcal{B} \subseteq S_1 \times S_2$ is a* bisimulation relation *if $(s_1, s_2) \in \mathcal{B}$ implies:*
- *$L_1(s_1) = L_2(s_2)$*
- *if $(s_1, s_1') \in R_1$, then $\exists s_2' \in S_2$ s.t. $(s_2, s_2') \in R_2$ and $(s_1', s_2') \in \mathcal{B}$*
- *if $(s_2, s_2') \in R_2$, then $\exists s_1' \in S_1$ s.t. $(s_1, s_1') \in R_1$ and $(s_1', s_2') \in \mathcal{B}$*

Lemma 2 (Correctness of Symmetry Reduction). *Given a transition system represented as a Kripke structure $\mathcal{K} = \langle S, I, R, L, AP \rangle$ and its quotient*

structure $\mathcal{K}_G = \langle S_G, I_G, R_G, L_G, AP \rangle$ *w.r.t. an invariance group* G *for all atomic propositions* $p \in AP$ *in symmetric LTL formulas* ϕ*, the structure* $\mathcal{K}_G$ *is* bisimulation equivalent *to* $\mathcal{K}$*.*

Theorem 2 (Correctness of SR Tendermint Model). *Given the Tendermint consensus protocol modelled as* $\mathcal{K} = \langle S, I, R, L, AP \rangle$ *and the reduced model* $\mathcal{K}_s = \langle S_s, I_s, R_s, L_s, AP \rangle$ *via symmetry reduction, if a counterexample exists in the state space of* $\mathcal{K}$ *then it also exists in the state space of* $\mathcal{K}_s$*.*

3.6 Other Notable Optimisations

Abstraction of Timed Components from the Initial TCSP# Model. Initially, we modelled the system using the RTS Module of the PAT model checker, which employs a global clock to impose a total order on events. However, the clock significantly increased model complexity and state space, causing the model checker to freeze. We then switched to the standard CSP# Module, removing the clock after realizing that event order could still be determined without exact timings. This shift led to a "result-oriented" approach, focusing on the outcomes of timed events like timeouts and deadlines while considering all possible interleavings. Eliminating the clock, which merely recorded timestamps, reduced the state space and simplified the model without compromising correctness.

Abstraction of Synchronization Primitives. The intended functions of certain artificially modelled synchronization primitives, such as distributed mutexes, which do not contribute to the core logical behaviour of the protocol, can be achieved more concisely using higher-level built-in constructs like unconditional choice operators. Since the unconditional choice operator inherently allows only one participating process to proceed, it naturally enforces mutual exclusion and prevents concurrent access to critical sections, eliminating the need for explicit synchronization. Following the "result-oriented" approach, these primitives can be abstracted away, avoiding manual steps like mutex acquisition and release while preserving the model's semantics. This simplification reduces process complexity and the number of possible interleavings, thus minimizing the state space.

Shared Use of A Single Instance of Message Log. Another optimization that may seem counter-intuitive is using a single message log for all processes, rather than giving each process its own log. Although individual logs might align more closely with the system's semantic interpretation, they are unnecessary and redundant in PAT. The primary concern — distinguishing the arrival order of a broadcast message across n processes (yielding $n!$ possible orders) — is effectively addressed by using a general choice operator. This operator randomly selects the next process to handle the broadcast message, ensuring that all possible execution traces are explored. Given that our system model assumes a partially synchronous network (where all messages are eventually delivered), the variability lies in timing rather than in message drops or corruption.

Property: safety + liveness						
Exp. No.	**Timeout Mech.**	**Byzantine**	**Reduction**		**Result**	**Time**
			POR	SR		
1	✓	×	×	×	CRASH	2 days
2	✓	×	✓	×	VALID	10 hours
3	✓	×	✓	✓	VALID	325.82s
4	✓	✓	✓	×	CRASH	2 days
5	✓	✓	✓	✓	VALID	242.59s
6	×	×	✓	✓	VALID	0.21s
7	×	✓	✓	✓	VALID	0.14s

Table 1: Verification results of our CSP# models for Tendermint. Specs: Intel Core i7 11800H / 32GB RAM.

Unlike the *constant-factor* improvements in *time* and *space* complexity achieved by abstracting synchronization primitives and using a single message log, abstracting timed components from our original TCSP# model leads to *exponential* state space reduction, from 4^n to 2^n, where n is the number of processes, for the decisions on timeout.

3.7 Discussions on Generalisability

Our proposed partial-order reduction techniques are also applicable to other BFT protocols because usually only a few dependencies between functions or procedures within the same process or across different processes can be observed (since if not, all steps are dependent on one another, forming a totally-ordered execution sequence, which obviously does not need to be further reduced), opening up the door to reduce these interleaved executions which are stuttering equivalent and therefore well-founded *linearizations* of the partial order execution.

Our symmetry reduction techniques can similarly be applied to exploit symmetries inherent in other typical BFT protocols which share a similar structure with respect to the concept of rounds and steps (or sometimes called phases) within a round (*i.e.*, both properties are orderable and hence can be sorted), and at the same time do not distinguish between roles undertaken by their participating nodes/processes and treat them as equal counterparts that perform exactly the set of rules or procedures at an appropriate time (*i.e.*, upon the stipulated pre-condition of each rule becomes true).

4 Verification Results

The verification results of our CSP# model built specifically for the Tendermint consensus algorithm were produced via the PAT model checker under different honesty configurations, as shown in Table 1. Note that before applying the reduction techniques, the verification used up all the RAM after 2 days.

In the verification, we check LTL formulae of the form

$$\Box\Diamond(decision[0] == T \wedge decision[1] == T \wedge \cdots)$$

to express both the safety property and the liveness property (cf. end of Section 2) — from any state ($\Box$), eventually ($\Diamond$), the decisions of node 0, 1, $\cdots$ will be made and will be the same (T). We examined all honest processes and cases where one process is Byzantine, applying our vulnerability fix. The results show that the algorithm maintains both *safety* and *liveness* even when a Byzantine process is present. The Byzantine process can reject valid proposals or make invalid ones, but its behaviour is limited to keep the state space manageable and ensure termination. Complex behaviours like *equivocation* and *amnesia* are excluded to simplify analysis.

From Table 1, we can observe that if we remove the timeout mechanism in the protocol (last two rows in Table 1) the verification time is significantly reduced. Timeouts significantly increase the state space complexity, as they add variability in message delays and process execution timing, which needs to be accurately modelled for realistic verification.

The successful verification of Tendermint's *safety* and *liveness* properties and the application of reduction techniques are notable contributions of this work. They represent a breakthrough in addressing the complexities of Byzantine fault-tolerant consensus verification, offering a more robust solution than existing verifications such as that in TLA+, which could not verify liveness due to limitations in managing state space.

These verification results prove the correctness of the Tendermint consensus algorithm at a single blockchain height. By induction, the correctness of the entire blockchain is established. Starting with the base case, the genesis block is inherently valid as the blockchain's founder creates it. For the inductive step, assuming correctness up to height k, the block at height $k + 1$ must also be correct when derived using this consensus algorithm. This completes the proof, ensuring both *safety* and *liveness* of the blockchain.

Also shown in Table 1, the only experiment (5) that finished verification in a reasonable time when considering both timeout mechanism and Byzantine nodes has to implement both of our proposed reduction techniques. Note that verification results about the impact of other notable optimizations in subsection 3.6 are omitted from Table 1, as their exponential improvement is asymptotically less effective in practice than the primary techniques discussed above.

5 Related Work

Blockchain Consensus. Numerous consensus protocols have been proposed for blockchain, as surveyed in [27]. Among them, BFT based consensus protocols form a large category. Compared to other protocols like PoW (proof-of-work) and PoS (proof-of-stake) which assign the participants with high computing power, stakes and storage, a higher priority in winning the leader competition, BFT-based allow participants to reach consensus through voting. Tendermint protocol analysed in this work is BFT-based in general, and their voting weight are decided following the idea of PoS which is beyond the scope of this work.

Formal Verification of Blockchain Consensus. Compared to the vast number of consensus protocols, only a few have been formally verified. Notable examples include the Snow White consensus for the Avalanche blockchain, proven using mathematical proof [16]; the Algorand consensus [3] and Raft consensus [26], verified using Coq; the Stellar consensus, verified with the model checker UPPAAL [28]; the Red Belly consensus, verified with ByMC [25]; and the Beacon Chain in Ethereum 2.0 and the Trust-based Blockchain Crowdsourcing (TBC) consensus protocol, verified with PAT [1,2]. Additionally, a branch of work focuses on formal verification logic and techniques for fault-tolerant distributed algorithms, verifying protocols like Last Voting, OneThirdRule, and Paxos [17,6], which ensure crash fault tolerance but lack the Byzantine fault tolerance.

Formal Verification of Tendermint. Notably the following:

Manual Analysis. The work [4] manually proved a few properties of Tendermint, including termination, validity, integrity and agreement of one-shot consensus, termination, validity and agreement of multiple rounds, and fairness, considering Byzantine nodes. Similar to this work, we also verified multiple rounds in the presence of Byzantine nodes. By contrast, we did not verify fairness (due to the required game theory verification being challenging for model checkers), and we used automated formal verification to reduce human errors in proofs.

The TLA+ Model. The work [22] models the Tendermint consensus protocol using TLA+, but does not incorporate any Byzantine behaviours. Properties like *equivocation* (*i.e.*, sending multiple messages that vote for different values within the same round to other processes) and *amnesia* (*i.e.*, locking a new value in a round without unlocking a previously locked value from some earlier round) are modelled and proven. However, the *safety* and *liveness* properties are excluded from the verification objectives, likely due to the set-theory-based characteristics inherent in the TLA+ specification language, which, while expressive and helpful in simplifying definitions through constructs such as universal and existential quantifiers, lacks the ability to enforce an execution order of upon rules among different processes. Under the same project, TLA+ has been used to verify other related protocols such as Fastsync—a synchronization mechanism to recover from network disconnection [7], which is beyond the analysis target of this work. The work [5] also used TLA+ to model Tendermint consensus, however, focusing on verifying a Nash Equilibrium property to identify free-riding.

The Ivy Model. The same project as in [22] has also provided an Ivy model of Tendermint consensus and proves the safety property by refinement of an abstract model with the actual model. However, the property is with respect to Byzantine nodes, which do not support sending forged messages.

The PAT model. Maung et al. [21] used PAT to analyse Tendermint consensus for safety properties BUT NOT liveness property. They verified a few attack scenarios for multiple rounds, but they used the old Tendermint specification, which suffers from liveness attacks. In contrast, this work verified the new version of Tendermint, proved its liveness, and identified other vulnerabilities.

Compared to previous works, we have successfully verified both *safety* and *liveness* properties using advanced reduction techniques, even in scenarios with either no Byzantine processes or exactly f Byzantine processes among $3f + 1$ total processes. This extends the state space coverage and accounts for a wider range of arbitrary Byzantine behaviors, making our verification results more robust and trustworthy.

6 Conclusion and Future Work

This work verified the new Tendermint consensus protocol in multiple rounds with Byzantine nodes. It advanced existing verification by enhancing the ability to verify liveness in multiple rounds using a timeout mechanism. The approach can be applied to other blockchain consensuses with similar state-space challenges. We found a vulnerability in Tendermint that could lead to a Denial of Service attack and proposed a fix to eliminate it. We optimized our model with reduction techniques to minimize the state space, resulting in an efficiently model-checked model. Our patched model meets both safety and liveness properties. Future work includes modelling a more complete set of Byzantine behaviours, scaling up the number of nodes in the model checking process, extending verification scope from specifications to actual implementations to examine behavioural consistency, and applying our reduction techniques to other BFT protocols towards a universal framework compatible with existing tools.

Data Availability. All versions of our models and detailed proofs are available at `https://tinyurl.com/4hbaf62m`. The PAT model checker can be downloaded at `https://pat.comp.nus.edu.sg/resources/pat3_5_1/`.

References

1. Afzaal, H., Imran, M., Janjua, M.U.: Formal verification of persistence and liveness in the trust-based blockchain crowdsourcing consensus protocol. Computer Communications **192**, 384–401 (2022)
2. Afzaal, H., Zafar, N.A., Tehseen, A., Kousar, S., Imran, M.: Formal verification of justification and finalization in beacon chain. IEEE Access **12**, 55077–55102 (2024)
3. Alturki, M.A., Chen, J., Luchangco, V., Moore, B., Palmskog, K., Peña, L., Roşu, G.: Towards a verified model of the algorand consensus protocol in coq. In: Formal Methods. FM 2019 International Workshops. pp. 362–367. Springer (2020)
4. Amoussou-Guenou, Y., Pozzo, A.D., Potop-Butucaru, M., Tucci-Piergiovanni, S.: Correctness and fairness of tendermint-core blockchains (2018), `https://arxiv.org/abs/1805.08429`
5. Baloochestani, A., Jehl, L.: Eiffel: Extending formal verification of distributed algorithms to utility analysis. In: 5th Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS). pp. 1–7 (2023)
6. Berkovits, I., Lazić, M., Losa, G., Padon, O., Shoham, S.: Verification of threshold-based distributed algorithms by decomposition to decidable logics. In: Computer Aided Verification CAV. pp. 245–266. Springer (2019)

7. Braithwaite, S., Buchman, E., Konnov, I., Milosevic, Z., Stoilkovska, I., Widder, J., Zamfir, A.: Tendermint blockchain synchronization: Formal specification and model checking. In: Leveraging Applications of Formal Methods (ISoLA) (2020)
8. Bride, H., Cai, C., Dong, J.S., Goré, R., Hóu, Z., Mahony, B.P., McCarthy, J.: N-PAT: A nested model-checker - (system description). In: The International Joint Conference on Automated Reasoning, IJCAR 2020. Springer (2020)
9. Buchman, E.: Tendermint: Byzantine fault tolerance in the age of blockchains (2016), https://api.semanticscholar.org/CorpusID:59082906
10. Buchman, E., Kwon, J., Milosevic, Z.: The latest gossip on BFT consensus. CoRR **abs/1807.04938** (2018), http://arxiv.org/abs/1807.04938
11. Castro, M., Liskov, B.: Practical byzantine fault tolerance. In: Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI). pp. 173–186 (1999)
12. Clarke, E.M., Emerson, E.A., Jha, S., Sistla, A.P.: Symmetry reductions in model checking. In: Computer Aided Verification, CAV '98. vol. 1427, pp. 147–158 (1998)
13. Clarke, E.M., Grumberg, O., Minea, M., Peled, D.A.: State space reduction using partial order techniques. Int. J. Softw. Tools Technol. Transf. **2**(3), 279–287 (1999)
14. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.): Handbook of Model Checking. Springer (2018)
15. Clarke, E.M., Jha, S., Enders, R., Filkorn, T.: Exploiting symmetry in temporal logic model checking. Formal Methods Syst. Des. **9**(1/2), 77–104 (1996)
16. Daian, P., Pass, R., Shi, E.: Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake. In: 23rd International Conference on Financial Cryptography and Data Security, FC. p. 23–41. Springer-Verlag (2019)
17. Drăgoi, C., Henzinger, T.A., Veith, H., Widder, J., Zufferey, D.: A logic-based framework for verifying consensus algorithms. In: 15th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI (2014)
18. Dwork, C., Lynch, N.A., Stockmeyer, L.J.: Consensus in the presence of partial synchrony. J. ACM **35**(2), 288–323 (1988)
19. Godefroid, P.: Exploiting symmetry when model-checking software. In: Formal Methods for Protocol Engineering and Distributed Systems (1999)
20. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
21. Maung Maung Thin, W.Y., Dong, N., Bai, G., Dong, J.S.: Formal analysis of a proof-of-stake blockchain. In: 23rd International Conference on Engineering of Complex Computer Systems (ICECCS). pp. 197–200 (2018)
22. Milosevic, Z., Konnov, I.: Tendermint consensus tla+ model. https://tinyurl.com/26j6vfd7 (2021)
23. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: towards flexible verification under fairness. In: Computer Aided Verification, CAV (2009)
24. Team, T.C.: What is tendermint, https://tinyurl.com/4ewab889
25. Tholoniat, P., Gramoli, V.: Formal Verification of Blockchain Byzantine Fault Tolerance, pp. 389–412. Springer International Publishing, Cham (2022)
26. Woos, D., Wilcox, J.R., Anton, S., Tatlock, Z., Ernst, M.D., Anderson, T.: Planning for change in a formal verification of the raft consensus protocol. In: ACM SIGPLAN Conference on Certified Programs and Proofs (2016)
27. Xu, J., Wang, C., Jia, X.: A survey of blockchain consensus protocols. ACM Comput. Surv. **55**(13s) (2023)
28. Yoo, J., Jung, Y., Shin, D., Bae, M., Jee, E.: Formal modeling and verification of a federated byzantine agreement algorithm for blockchain platforms. In: IEEE International Workshop on Blockchain Oriented Software Engineering (2019)