Formal Analysis of a Proof-of-Stake Blockchain

Wai Yan Maung Maung Thin[1], Naipeng Dong[1], Guangdong Bai[2], and Jin Song Dong[1]

[1] National University of Singapore
[2] Singapore Institute of Technology

Abstract. Blockchain technology relies on consensus algorithms to resolve conflicts in Byzantine environments. New blockchain algorithms are rapidly designed and implemented without a properly conducted formal analysis and verification. In this paper, we conducted a study on Tendermint which is a proof-of-stake consensus algorithm. We verified that the consensus protocol is deadlock-free and is able to reach consensus when at least 2/3 of the network is in agreement. We also proved that a minority set of nodes that compose more than 1/3 of the network is enough to censor the majority of the network and prevent the network from reaching consensus and conclude that the algorithm has some shortcomings on availability.

Keywords: formal verification, blockchain, tendermint, consensus, proof-of-stake

1 Introduction

Introduced by Bitcoin in 2008 as a means to provide a trustworthy service among non-trusting peers without a central governing body, the blockchain technology has long evolved from its traditional use as a decentralized ledger for digital currencies. Many platforms such as Ethereum [2] and Ripple [7] are already using blockchains to support smart contracts which allow the execution of arbitrary codes in a decentralized manner [11]. The technology can also be easily applied to several fields such as Internet of Things (IoT) [12] or applications such as data privacy tools [13] and reputation systems [8].

The growth of the ecosystem and the adoption of the blockchain technology in various domains and applications have created a competitive environment where newer protocols and algorithms are being rapidly introduced and implemented without a formal analysis and verification process. The lack of related published research work implies that many of the widely used protocols and algorithms have yet to undergo a formal analysis that verifies the security properties and the correctness of the system. This process is extremely important, especially for the consensus algorithms that mandate how to resolve conflicting claims in the absence of a central authority and define the consistency, performance, scalability and liveness of the system.

Depending on the nature of the blockchain, the focus of the consensus algorithm is also different. For instance, Tendermint [6] favours consistency over liveness (availability) whereas Casper [4], the consensus algorithm of Ethereum sacrifices consistency to achieve plausible liveness. Previous research works on Tendermint blockchain [1] did not focus on the formal analysis of the consensus protocol. This paper aims to fill the gap by providing a formal analysis of the Tendermint consensus protocol in CSP# [9] using the PAT [10] model checker. We prove that the consensus protocol is deadlock-free and is able to reach consensus when at least 2/3 of the network is in agreement.

2 Background

2.1 Blockchain Architecture

A blockchain is a sequence of blocks, each of which maintains the hash value of itself and the link to its previous block (with the exception of the *genesis* block which is the first block in the chain). This design architecture ensures that the contents for an existing block (for example, transactions in a block) in the chain cannot be updated without modifying the hash values of the following blocks. Contents of a block widely vary based on the application. A typical block consists of a block header and a block body. The header is used to store the metadata of the previous and the current block, such as the signatures and the hash values. The body is used as a payload and can be used to store anything, ranging from messages and transactions to smart contracts.

A blockchain network is a set of nodes responsible for keeping track of the blockchain and validating and appending new blocks to the blockchain. Based on the difference in functionality, nodes can be categorized into two - miners and validators. Miners store only the last few blocks of the blockchain and are responsible for creating new blocks. Validators store a full copy of the blockchain and ensure that the new block submitted by the miners is valid and conforms to the rules of the blockchain. Based on the architecture of the blockchain, it is possible for a node to be both a miner and a validator at the same time.

2.2 Consensus in Blockchains

The nodes in the blockchain network operate under the fact that there is no central authority governing over them. In a perfect scenario, all the members of the network always agree on the same new block to be appended to the blockchain and there exists only a single blockchain in the whole network. In reality, however, nodes might get disconnected from the network or even act maliciously in Byzantine environments on purpose. Thus, a fault-tolerant consensus protocol is required which is agreed by all the nodes in the blockchain to resolve any potential conflict.

Consensus protocols differ based on the aspects or the attributes of the node that are being assessed. Proof-of-work (PoW) requires the node to provide the computing power to solve a mathematical problem in order to append a new block to the blockchain. The most well-known user of PoW consensus is Bitcoin where miners have to solve cryptographic hash puzzles as proof of work. In Proof-of-stake (PoS), a node has to stake something it owns, usually in the form of a cryptocurrency. If a malicious node tries to manipulate the blockchain and the other nodes detect it, the locked-up stakes get slashed or rewards are withheld. Delegated proof-of-stake (DPoS) is where the nodes of the network vote for a set of nodes to be the delegators.

In a PoW system, the nodes are rewarded for performing an operation that is agreed by a majority of the nodes in the system. The caveat here is that participants are not punished for performing a malicious operation. As a result, PoW systems cannot deter the participants from performing a selfish mining [5] or participating in a 51% attack. In order to solve this problem, newer generations of blockchains (Ethereum, Tendermint, etc.) have started to use proof-of-stake as the consensus algorithm. In a PoS system, the participants are rewarded for performing a non-malicious activity just like a PoW system but they are also held accountable and are punished for any malicious operation.

2.3 Modeling Language

Our Tendermint consensus model is built using CSP# [9] and is verified using PAT [10] model checker. CSP# integrates the high-level modeling operators of CSP with low-level procedural codes in C# language and supports custom data structures which is extensively used in our model to represent the properties and the behaviors of the blocks and the blockchain. The following is a list of CSP# syntax and semantics commonly used in modelling concurrent systems.

Process $P ::= Stop$	– deadlock
$\mid Skip$	– termination
$\mid [b]P$	– state guard
$\mid e \rightarrow P$	– event prefixing
$\mid e\{program\} \rightarrow P$	– data operation prefixing
$\mid c?d \rightarrow P(d)$	– channel input
$\mid c!d \rightarrow P$	– channel output
$\mid P;\ Q$	– sequence
$\mid P \sqcap Q$	– internal choice
$\mid P \Box Q$	– external choice
$\mid if\ \ b\ \ then\ \ P\ \ else\ \ Q$	– conditional branch
$\mid P \mid\mid Q$	– synchronous
$\mid P \mid\mid\mid Q$	– asynchronous

where P and Q are *processes*, b is a condition, e is a simple event, *program* is a block of code that is atomically executed and c is a synchronized communication channel.

CSP# uses a Labeled Transition System (LTS) for model checking. This is represented as a tuple (S, S_{init}, A, T) where S is a finite set of states, $S_{init} \in S$ is the initial state, A is a set of actions and T is a set of labeled transition relations. T is a set of tuples (s, e, s') where event e causes the system to change its state from s to s'. We say a state s_n is reachable if there exists a path P such that $\forall\, i < n;\ (s_i, e_i, s'_{i+1}) \in T \wedge s_0 = S_{init}$. We will be using this reachability checking to prove the security properties of the consensus algorithm.

3 Tendermint

Tendermint is a proof-of-stake consensus protocol that is Byzantine fault tolerant. Participants in the protocol are called validators. There is no concept of miners in Tendermint and thus, validators are also responsible for the creation of new blocks. The height of the chain increases every time a block is added to the chain. Validators are chosen in a round-robin manner to become the proposer who is in charge of creating and proposing a block for the current round. Validators are required to post a *bond transaction* that will lock a set amount of his coins (stakes) for a set duration. If the validator is found to be involved in any malicious activity within this duration, it can be punished by slashing away its deposited stake. After this duration, stakes is unlocked and returned to the validator.

3.1 Consensus Algorithm

The consensus algorithm consists of five steps - *Propose*, *Prevote*, *Precommit*, *Commit* and *NewHeight*.

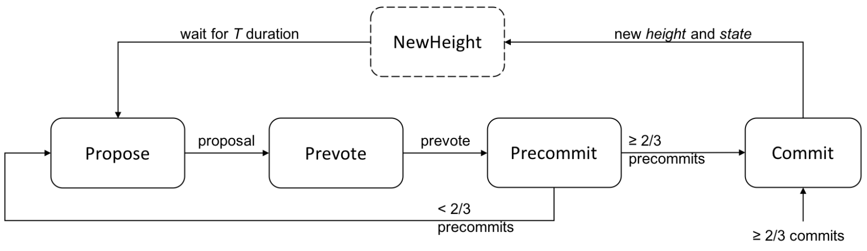

Fig. 1. State machine of the consensus protocol, adapted from *Tendermint: Consensus without mining* [6]

In the *Propose* step, the proposer broadcasts a proposal to its peers. A proposal includes the block, the signatures of the validators who have validated the block, the signature of the proposer as well as the round and the height information. If the proposer has already locked on a block during the precommit of the previous round, that block will be used for proposal. Otherwise, a new block will be created. All nodes will gossip the *proposal* to their neighbouring peers during this period.

In the *Prevote* step, each validator will vote for a block and gossip it to the neighbours. A vote consists of the hash of the voted block, the signature of the voter, type of vote - whether it is a *prevote* or a *precommit* plus the round and the height information. The block to be included is chosen in the following order - (1) a locked proposed block from prior rounds and (2) a valid acceptable block from the current proposal. If neither is available, a special NIL prevote is broadcast to the neighbours. All nodes will gossip all *prevotes* for the round to their neighbouring peers.

In the *Precommit* step, the validator checks if it has received more than 2/3 of prevotes for an acceptable block. If there is one, the validator releases the existing lock, instead locks onto this block and signs and broadcasts a precommit vote for this block. The validator also packages the prevotes for the locked block into a proof-of-lock which will be used to create the block in the next *Proposal*. In the case where there are fewer than 2/3 prevotes, the validator will neither sign nor lock on any block. During this period, all nodes will gossip all *precommits* for the round to all neighbouring peers. At the end of *Precommit*, if the node has received more than 2/3 of precommits for a particular block, it will proceed to *Commit* step. Otherwise, it transits to the *Propose* step of the next round.

In the *Commit* step, two parallel conditions must be fulfilled before the consensus algorithm can cycle back to *Propose* step. First and foremost, the node must have received the block from one of its peers so that it can sign and broadcast a commit to other peers. Second, the node must wait until at least 2/3 commits of the block are received by the network. Once these are satisfied, the node will set the *CommitTime* to current time and move on to *NewHeight* step where the nodes will stay for a fixed duration. The purpose is to allow the nodes to wait for additional commits of the committed block which were not received in *Precommit* due to network latency issues. After the set duration is up, the algorithm starts again from *Propose*. At anytime during the consensus process, if a node receives more than 2/3 commits for a particular block, it will immediately enter the *Commit* step.

4 Modelling

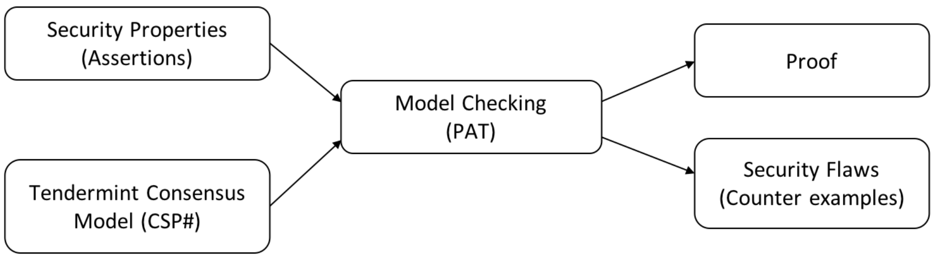
Security Properties (Assertions)
Tendermint Consensus Model (CSP#)
Model Checking (PAT)
Proof
Security Flaws (Counter examples)

Fig. 2. Overview of the verification approach

Figure 2 shows the overview of the verification approach. We first implement the Tendermint consensus model in CSP# and define its security properties. We also define a set of assertion rules to capture the security properties that we want to verify against the model. We run the simulation through PAT which either gives us verification of satisfied assertions or the traces of the unsatisfied assertions.

For the purpose of this paper, we focused on the consensus protocol of the blockchain and as such we have abstracted the structure and the content of the block. We modelled the channel communication among nodes only for consensus. Data exchange among the nodes for the internal contents of the blocks such as transactions and contracts is considered to be outside of the scope of this work. We ran two sets of verification - one with 3 validators in the network and the other with 4 validators (where 1 node represents $< 1/3$ of the network; 2 nodes represent $\geqslant 1/3$ and $\leqslant 2/3$ of the network and; 3 nodes represent $> 2/3$ of the network). The number of nodes in the network is limited to 4 to reduce the number of states in the system. This is the minimum amount of nodes required to prove the properties of the model while keeping the states small enough for PAT to verify. The following assumptions have been made in the model.

1. All the nodes in the network are connected to each other.
2. Existing nodes will not leave the network and no new nodes will join the network.
3. All nodes have the same weight or voting power.
4. There is no latency in the network.

First, we define a set of variables used in the system.

```
#define N 4;
#define MINORITY 1;
#define HALF 2;
#define MAJORITY 3;
#define R 2;

#define HONEST_VOTING 0;
#define OVERWRITE_VOTING 1;
#define INVALID_BLOCK_VOTING 2;
#define NO_VOTING 3;

var proposer = 0;
var round = 1;
var votingBehaviour[N];
channel c01 0;
.
.
channel c32 0;
```

where N is the number of nodes in the network and R stands for the number of rounds to be simulated. *proposer* is the proposer chosen by round-robin for *round*. *votingBehaviour* is used to set the behavior of the nodes in the system which can be one of the following values: $HONEST\_VOTING$, $OVERWRITE\_VOTING$, $INVALID\_BLOCK\_VOTING$, NO_VOTING. *channel* c_{ij} represents a synchronized communication channel from node i to node j.

4.1 Blockchain

We define a blockchain as a set of nodes N where each node $x \in N$ executes a sequence of processes in parallel. Since there is no network latency in our model, *NewHeight* is not required and *Commit* does not need to wait until the node receives the block. We also split the *PreparePOL* from *Precommit* since preparing proof-of-lock happens at the end of *Precommit* and not during. At the end of each round, the *NextRound*() process is executed to select the next proposer.

$$\mathit{BlockChain}() = (|| \; x : \{0..N-1\} \bullet (\mathit{Propose}(x);\ \mathit{Prevote}(x);\ \mathit{Precommit}(x);\ \mathit{PreparePOL}(x);\ \mathit{Commit}(x)));\ \mathit{NextRound}();$$

where $P;\ Q$ represents process P followed by process Q and $P \; || \; Q$ represents synchronous processes P and Q.

4.2 Propose

The *Propose*(x) process is defined as follows. If a node is a *proposer* for this round, it will broadcast the proposal to other nodes using the communication channels. Otherwise, it will wait for a proposal from the proposer.

```
Propose(i) =
    [proposer == i] proposeBlock.i {
        proposedBlock = lockedBlockList.Get(i);
        if(proposedBlock.GetHash() == -1) {
                proposedBlock = new Block(i);
        }
        proposalList.Set(i, new Proposal(proposedBlock, new
            Signature(i)))
    } -> ((
        [i == 0] (
             c01!proposalList.Get(i) -> Skip ||
                c02!proposalList.Get(i) -> Skip ||
                c03!proposalList.Get(i) -> Skip
        ) []
        [i == 1] (
            c10!proposalList.Get(i) -> Skip ||
            c12!proposalList.Get(i) -> Skip ||
            c13!proposalList.Get(i) -> Skip
        ) []
        [i == 2] (
            c20!proposalList.Get(i) -> Skip ||
            c21!proposalList.Get(i) -> Skip ||
            c23!proposalList.Get(i) -> Skip
        ) []
        [i == 3] (
            c30!proposalList.Get(i) -> Skip ||
            c31!proposalList.Get(i) -> Skip ||
            c32!proposalList.Get(i) -> Skip
    )); Skip)
        []
        [i == 0] (
            c10?y -> {proposalList.Set(i, y)} -> Skip []
            c20?y -> {proposalList.Set(i, y)} -> Skip []
            c30?y -> {proposalList.Set(i, y)} -> Skip)
        []
        [i == 1] (
            c01?y -> {proposalList.Set(i, y)} -> Skip []
            c21?y -> {proposalList.Set(i, y)} -> Skip []
            c31?y -> {proposalList.Set(i, y)} -> Skip)
        []
        [i == 2] (
            c02?y -> {proposalList.Set(i, y)} -> Skip []
            c12?y -> {proposalList.Set(i, y)} -> Skip []
            c32?y -> {proposalList.Set(i, y)} -> Skip)
        []
        [i == 3] (
            c03?y -> {proposalList.Set(i, y)} -> Skip []
            c13?y -> {proposalList.Set(i, y)} -> Skip []
            c23?y -> {proposalList.Set(i, y)} -> Skip
);
```

4.3 Prevote

The *Prevote*(x) process is a sequence of two processes, prefixed by event *propose_end* which is used to represent the end of the *Propose* step. The first process *Prevote_x*() validates the proposal received from *Propose*(x) and votes accordingly. The second process *BroadcastPrevotes*() represents the broadcasting and the receiving of votes among the nodes. A node will not broadcast the vote if the *censor* flag is active. The definition of this process is slightly different for each node since different channels are used for communication. However, the logical behavior is the same for every node and has been marked with "...." below for simplicity.

```
Prevote(i) =
        [i == 0] propose_end -> Prevote_0(); (
        (
          [censor0 == 0] (
              c01!tmpVote0 -> Skip ||
              c02!tmpVote0 -> Skip ||
              c03!tmpVote0 -> Skip)
          []
          [censor0 == 1] Skip
        )
        ||
        (
          [censor1 == 0] c10?y { prevote0.Add(y) } -> Skip []
```

```
    [censor1 == 1] Skip
  )
  ||
  (
    [censor2 == 0] c20?y { prevote0.Add(y) } -> Skip []
    [censor2 == 1] Skip
  )
  ||
  (
    [censor3 == 0] c30?y { prevote0.Add(y) } -> Skip []
    [censor3 == 1] Skip
  ))
  []
  [i == 1] propose_end -> Prevote_1(); ....
  []
  [i == 2] propose_end -> Prevote_2(); ....
  []
  [i == 3] propose_end -> Prevote_3(); ....
```

4.4 Prevote

The malicious node behavior is simulated in process *Prevote_x*(). It validates the proposal received during *Propose* and broadcasts a vote accordingly. An honest node validates the block as is whereas a malicious node with the intent to overwrite the proposed block broadcasts a different block from the one it received. A malicious node with the intent to vote an invalid block is simulated as voting for a duplicate block that already exists in the chain. A node who wants to censor the proposer is marked by a flag *censor_i*. *Prevote_*1(), *Prevote_*2() and *Prevote_*3() share the same logical behavior with *Prevote_*0() and will not be listed for the sake of simplicity.

```
Prevote_0() = validateProposal.0 {
    tmpProposedBlock0 = lockedBlockList.Get(0);
    if(tmpProposedBlock0.GetHash() == -1) {
        tmpProposal0 = proposalList.Get(0);
        tmpProposedBlock0 = tmpProposal0.GetBlock();
        var invalid = chain0.Contains(tmpProposedBlock0);
        if(invalid) {
            tmpProposedBlock0 = new Block();
        }
    }
    if(votingBehaviour[0] == HONEST_VOTING) {
        tmpVote0 = new Vote(tmpProposedBlock0.GetHash(),
            signature0);
    } else if(votingBehaviour[0] == OVERWRITE_VOTING) {
        tmpVote0 = new Vote(1000, signature0);
    } else if(votingBehaviour[0] == INVALID_BLOCK_VOTING) {
        tmpProposedBlock0 = chain0.Peek();
```

```
          tmpVote0 = new Vote(tmpProposedBlock0.GetHash(),
              signature0);
      } else if(votingBehaviour[0] == NO_VOTING) {
          censor0 = 1
      }
      prevote0.Add(tmpVote0);
} -> Skip;
```

4.5 Precommit

The *Precommit*(x) process is similar to *Prevote*(x), defined by another sequence of two processes. The first process *Precommit_x*() retrieves the first block with at least 2/3 confidence from the votes received from *Prevote*(x) step. The second process *BroadcastPrecommits*() is similar to *BroadcastPrevotes*().

```
Precommit(i) =
          [i == 0] prevote_end -> Precommit_0(); ... []
          [i == 1] prevote_end -> Precommit_1(); ... []
          [i == 2] prevote_end -> Precommit_2(); ... []
          [i == 3] prevote_end -> Precommit_3(); ... ;

Precommit_0() = validateVote.0 {
          tmpProposedBlock0 =
              prevote0.GetFirstBlockWithMinSupport(MAJORITY);
          lockedBlockList.Set(0, tmpProposedBlock0);
          tmpVote0 = new Vote(tmpProposedBlock0.GetHash(),
              signature0);
          precommit0.Add(tmpVote0);
} -> Skip;
```

4.6 Prepare Proof of Lock

The *PreparePOL*(x) process prepares the proof of lock, gathers the signatures of the validators who voted for this block and stores them inside the block.

```
PreparePOL(i) =
          [i == 0] precommit_end -> prep_proof_of_lock.0 {
            tmpProposedBlock0 = lockedBlockList.Get(0);
            sigList0 = precommit0.
                getSignaturesForBlock(tmpProposedBlock0.GetHash());
            tmpProposedBlock0.SetSignatureList(sigList0);
          } -> polCompleted -> Skip
          []
          [i == 1] precommit_end -> prep_proof_of_lock.1 { .. }
              -> polCompleted -> Skip
          []
          [i == 2] precommit_end -> prep_proof_of_lock.2 { .. }
              -> polCompleted -> Skip
          []
```

```
        [i == 3] precommit_end -> prep_proof_of_lock.3 { .. }
            -> polCompleted -> Skip;
```

4.7 Commit

The *Commit*(x) process adds the precommitted block having at least 2/3 consensus to the chain. The node adds the block to the chain only if it is not *NIL*. This marks the end of one round of consensus.

```
Commit(i) =
        [i == 0] addtoChain.i {
                tmpProposedBlock0 =
                    precommit0.GetFirstBlockWithMinSupport(MAJORITY);
                if(tmpProposedBlock0.GetHash() != -1) {
                        chain0.Add(tmpProposedBlock0)
                }
        } -> Skip
        []
        [i == 1] addtoChain.i { .. } -> Skip
        []
        [i == 2] addtoChain.i { .. } -> Skip
        []
        [i == 3] addtoChain.i { .. } -> Skip;
```

5 Properties

We aim to verify the following set of properties that are claimed.

5.1 Deadlockfree-ness

A deadlock-free model means that at any point in time, no node should be waiting for another node to broadcast or receive a proposal or a vote, continue with validating of a block, or committing a block to its chain, even in Byzantine environments. We can use PAT to query for the deadlocks in the system.

```
#assert BlockChain deadlockfree
#assert BlockChainWithMinorityOverwrite() deadlockfree;
#assert BlockChainWithHalfOverwrite() deadlockfree;
#assert BlockChainWithMajorityOverwrite() deadlockfree;
#assert BlockChainWithMinorityInvalid() deadlockfree;
#assert BlockChainWithHalfInvalid() deadlockfree;
#assert BlockChainWithMajorityInvalid() deadlockfree;
#assert BlockChainWithMinorityCensor() deadlockfree;
#assert BlockChainWithHalfCensor() deadlockfree;
#assert BlockChainWithMajorityCensor() deadlockfree;
```

5.2 Ability to reach consensus

All the nodes in the network must be able to agree on one block and commit the same block to their own copy of the chain at the end of a round.

```
#define Consensus (!chain0.IsEmpty() && chain0.Peek() ==
    chain1.Peek() && chain1.Peek() == chain2.Peek() &&
    chain2.Peek() == chain3.Peek());

#assert BlockChain() reaches Consensus;
#assert BlockChainWithMinorityOverwrite() reaches Consensus;
#assert BlockChainWithHalfOverwrite() reaches Consensus;
#assert BlockChainWithMajorityOverwrite() reaches Consensus;
#assert BlockChainWithMinorityInvalid() reaches Consensus;
#assert BlockChainWithHalfInvalid() reaches Consensus;
#assert BlockChainWithMajorityInvalid() reaches Consensus;
#assert BlockChainWithMinorityCensor() reaches Consensus;
#assert BlockChainWithHalfCensor() reaches Consensus;
#assert BlockChainWithMajorityCensor() reaches Consensus;
```

5.3 Overwriting proposed block

This is an attack where a set of nodes collude to reject the proposed block and propose a new block of their own. For a blockchain to be resistant to this attack, all nodes must agree on the same block to be added to the chain after reaching consensus and the block must be identical to the originally proposed block.

```
#define ImmuneToOverwrite (round > 1 && !Consensus) ||
    (Consensus && proposedBlock == chain0.Peek());

#assert BlockChain() reaches ImmuneToOverwrite;
#assert BlockChainWithMinorityOverwrite() reaches
    ImmuneToOverwrite;
#assert BlockChainWithHalfOverwrite() reaches
    ImmuneToOverwrite;
#assert BlockChainWithMajorityOverwrite() reaches
    ImmuneToOverwrite;
```

5.4 Invalid blocks

The network must be able to reject invalid blocks submitted by malicious nodes that do not conform to the rules of the blockchain. This is simulated by malicious nodes trying to add a duplicate (existing) block to the chain.

```
#define ImmuneToInvalidBlockInsertion (round>1 &&
    !chain0.ContainsDuplicates() &&
    !chain1.ContainsDuplicates() &&
    !chain2.ContainsDuplicates() &&
    !chain3.ContainsDuplicates());

#assert BlockChain() reaches ImmuneToInvalidBlockInsertion;
#assert BlockChainWithMinorityInvalid() reaches
    ImmuneToInvalidBlockInsertion;
#assert BlockChainWithHalfInvalid() reaches
    ImmuneToInvalidBlockInsertion;
#assert BlockChainWithMajorityInvalid() reaches
    ImmuneToInvalidBlockInsertion;
```

5.5 Censorship attacks

Censorship attack is an attack where malicious nodes in the network refuse to broadcast or vote a valid block in order to censor a particular content of the block or censor the node itself. The blockchain is immune to this attack if other nodes can reach consensus even with the absence of malicious nodes in the voting process. The assertions used to check for the reachability of consensus can be reuse for this verification.

```
#assert BlockChainWithMinorityCensor() reaches Consensus;
#assert BlockChainWithHalfCensor() reaches Consensus;
#assert BlockChainWithMajorityCensor() reaches Consensus;
```

6 Results and Analysis

We will be verifying the properties of the model under normal conditions as well as against specific attacks in Byzantine environments. We define nine more variations of *Blockchain*() with different percentage of malicious nodes in the system with different threat factors.

```
BlockChainWithMinorityOverwrite() =
    SimulateMalicious(MINORITY, OVERWRITE_VOTING);
    BlockChain();
BlockChainWithHalfOverwrite() = SimulateMalicious(HALF,
    OVERWRITE_VOTING); BlockChain();
BlockChainWithMajorityOverwrite() =
    SimulateMalicious(MAJORITY, OVERWRITE_VOTING);
    BlockChain();
BlockChainWithMinorityInvalid() = SimulateMalicious(MINORITY,
    INVALID_BLOCK_VOTING); BlockChain();
BlockChainWithHalfInvalid() = SimulateMalicious(HALF,
    INVALID_BLOCK_VOTING); BlockChain();
BlockChainWithMajorityInvalid() = SimulateMalicious(MAJORITY,
    INVALID_BLOCK_VOTING); BlockChain();
BlockChainWithMinorityCensor() = SimulateMalicious(MINORITY,
    NO_VOTING); BlockChain();
```

```
BlockChainWithHalfCensor() = SimulateMalicious(HALF,
    NO_VOTING); BlockChain();
BlockChainWithMajorityCensor() = SimulateMalicious(MAJORITY,
    NO_VOTING); BlockChain();
```

6.1 General properties

Table 1 presents verification results of properties of the model in normal and Byzantine environments. The model is **deadlock-free** (**T1**) at all times but does not always reach **consensus** (**T2**). It is important to note that not being able to reach consensus does not always mean the system is vulnerable. Based on the threat model, it might be better for the network to agree on no block than to agree on the manipulated block. As such, consensus should be analyzed with respect to each individual threat model

Table 1. Analysis results for deadlock-freeness and consensus reachability

	Deadlock-free	**Consensus**
BlockChain	✓	✓
BlockChainWithMinorityOverwrite ($\leqslant 1/3$)	✓	✓
BlockChainWithHalfOverwrite ($> 1/3$ and $< 2/3$)	✓	✗
BlockChainWithMajorityOverwrite ($\geqslant 2/3$)	✓	✓
BlockChainWithMinorityInvalid	✓	✓
BlockChainWithHalfInvalid	✓	✗
BlockChainWithMajorityInvalid	✓	✗
BlockChainWithMinorityCensor	✓	✓
BlockChainWithHalfCensor	✓	✗
BlockChainWithMajorityCensor	✓	✗

6.2 Against proposed block overwrites

From Table 2, we can see that process *BlockChain*, *BlockChainWithMinorityOverwrite* and *BlockChainWithHalfOverwrite* are immune to overwrites. We have also learnt from Table 1 that *BlockChainWithHalfOverwrite* failed to reach consensus. By failing to reach consensus, the network did not commit a block during this round. This is in contrast to *BlockChainWithMajorityOverwrite* where the manipulated block voted by the majority ($\geqslant 2/3$) is added to the network instead of the proposed block.

Table 2. Analysis results for immunity to proposed block overwrite

	Immunity to block overwrite
BlockChain	✓
BlockChainWithMinorityOverwrite ($\leqslant 1/3$)	✓
BlockChainWithHalfOverwrite ($> 1/3$ and $< 2/3$)	✓
BlockChainWithMajorityOverwrite ($\geqslant 2/3$)	✗

6.3 Against invalid block insertions

All four variations of the *BlockChain* can withstand the invalid block insertions. The only difference is that *BlockChainWithMinorityInvalid* and *BlockChain* are able to reach consensus whereas the other two are not. *BlockChainWithMinorityInvalid* is able to insert a valid block to the chain but no block is inserted to the chain in the case of the other two.

Table 3. Analysis results for immunity to invalid block insertions

	Immunity to invalid block
BlockChain	✓
BlockChainWithMinorityInvalid ($\leqslant 1/3$)	✓
BlockChainWithHalfInvalid ($> 1/3$ and $< 2/3$)	✓
BlockChainWithMajorityInvalid ($\geqslant 2/3$)	✓

6.4 Against censorship attacks

As for the censorship attacks, we find that the blockchain becomes vulnerable once the number of malicious nodes increases past 1/3. We have stated before that it takes $\geqslant 2/3$ of the network to collude to vote on a same non-proposed block to add it to the blockchain. However, if the intent is not for overwriting but just for censorship or disruption of the service, this can be achieved with merely $> 1/3$ of the network just by having them not participating in the voting process. It is also possible that nodes get disconnected from the network without any malicious intent. In both cases, the chain will be prevented from adding new blocks until $\geqslant 2/3$ of the network participates in the voting. Since the proposer is chosen by round robin, it is possible for the malicious minority nodes to stay offline until it is their turn to vote or until the next *Propose* step begins, effectively censoring the majority of the network from publishing anything.

Table 4. Analysis results for immunity to censorship

	Immunity to censorship attacks
BlockChainWithMinorityCensor ($\leqslant 1/3$)	✓
BlockChainWithHalfCensor ($> 1/3$ and $< 2/3$)	✗
BlockChainWithMajorityCensor ($\geqslant 2/3$)	✗

7 Conclusions

In this paper, we discussed the need for the formal analysis and verification of blockchain protocols. We formally modelled a simple consensus algorithm for a proof-of-stake blockchain based on Tendermint and verified that the algorithm is deadlock free. We also proved that it takes $\geqslant 2/3$ of the network to reach consensus and it takes $> 1/3$ of the network to censor the majority of the nodes from publishing a new block. It is possible to update the algorithm that chooses the proposer to ignore disconnected nodes in the previous round but this will only mitigate the problem. Alternative solutions to this are to implement a time lock consensus protocol [3] or to punish the nodes that are offline regardless of their intent, similar to how a node is punished by participating in malicious voting in proof-of-stake.

References

1. Buchman, E.: Tendermint: Byzantine fault tolerance in the age of blockchains. Ph.D. thesis, The University of Guelph (2016)
2. Buterin, V.: Ethereum white paper. GitHub repository (2013)
3. Buterin, V.: The problem of censorship (2015), https://blog.ethereum.org/2015/06/06/the-problem-of-censorship/
4. Buterin, V., Griffith, V.: Casper the friendly finality gadget. arXiv preprint arXiv:1710.09437 (2017)
5. Eyal, I., Sirer, E.G.: Majority is not enough: Bitcoin mining is vulnerable. In: International conference on financial cryptography and data security. pp. 436–454. Springer (2014)
6. Kwon, J.: Tendermint: Consensus without mining. Retrieved May **18**, 2017 (2014)
7. Schwartz, D., Youngs, N., Britto, A.: The ripple protocol consensus algorithm. Ripple Labs Inc White Paper **5** (2014)
8. Sharples, M., Domingue, J.: The blockchain and kudos: A distributed system for educational record, reputation and reward. In: European Conference on Technology Enhanced Learning. pp. 490–496. Springer (2016)
9. Sun, J., Liu, Y., Dong, J.S., Chen, C.: Integrating specification and programs for system modeling and verification. In: Third IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE). pp. 127–135. IEEE (2009)
10. Sun, J., Liu, Y., Dong, J.S., Pang, J.: Pat: Towards flexible verification under fairness. In: International Conference on Computer Aided Verification. pp. 709–714. Springer (2009)
11. Swan, M.: Blockchain: Blueprint for a new economy. O'Reilly Media, Inc. (2015)
12. Zhang, Y., Wen, J.: An IoT electric business model based on the protocol of bitcoin. In: 18th International Conference on Intelligence in Next Generation Networks (ICIN). pp. 184–191. IEEE (2015)
13. Zyskind, G., Nathan, O., Pentland, A.S.: Decentralizing privacy: Using blockchain to protect personal data. In: Security and Privacy Workshops (SPW). pp. 180–184. IEEE (2015)