Veridise. Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

DogeChain

Veridise Inc.
May 6, 2022

► **Prepared For:**

DogeChain.
dogechain.info

► **Prepared By:**

Ben Mariano
Chaofan Shou
Jacob Van Geffen
Andreea Buterchi
Bryan Tan
Yu Feng
Isil Dillig

► **Contact Us:** contact@veridise.com

► **Version History:**

May 20, 2022	V1
May 5, 2022	Draft

Contents

1 Executive Summary

From May 1 to May 20, 2022, DogeChain engaged Veridise to review the security of the DogeChain consensus mechanism. The review covered block validation, signature generation and verification, JsonRPC, helpers, update mechanisms, node resumption, and distributed key generation. Veridise conducted this assessment over 15 person-weeks, with five engineers working from commit 9dc1491 of the `dogechain-lab/jury` and `./jury-main-contract` repository. The auditing strategy involved tool-assisted analysis of the source code performed by Veridise engineers. The tools that were used in the audit included a combination of static analysis, fuzzing, property-based testing, bounded model checking, and formal verification. Some of these tools were developed specifically for the purpose of performing a thorough audit of the DogeChain protocols and contracts.

Summary of issues detected. The audit uncovered 29 issues, 15 of which are assessed to be high severity by Veridise auditors. One of the high-severity issues (V-DOGE-VUL-001) involves missing input validation, which could lead to memory exhaustion. The issue could be leveraged by a malicious node operator to launch denial-of-service attacks against other nodes on the same subnet. The second high-severity issue (V-DOGE-VUL-002) is related to validators in the set listening on gossips from other validators to form consensus. Developers should assume that these gossips could be from a malicious validator in the network and properly sanitize the incoming messages in the channel. However, the current implementation does not fully validate and sanitize these unsafe messages.

V-DOGE-VUL-003 is another issue of high severity. The transaction pool module sets a threshold on the amount of transactions that could be kept inside the transaction pool (i.e., having more transactions than threshold makes the transaction pool `overflow`). The threshold is tracked by using a counter inside the `slotGauge` interface. When the counter is equal to the threshold, no new transactions would be added to the transaction pool. By flooding the JSON RPC interface with a significant amount of valid transactions (e.g., 40960 transactions when threshold is 4096), it is possible to make the counter of `slotGauge` equal to the threshold while never getting decremented, even when the transaction pool is empty later. This bug could cause the transaction pool to get locked and throw `ErrTxPoolOverflow` exceptions for all following new transactions.

Code assessment. The core DogeChain consensus protocol is based on IBFT, another Byzantine fault-tolerant protocol whose implementation (in Golang) is originally forked from *Polygon-edge*. At a high level, Byzantine consensus is achieved deterministically as follows: 1) a leader or bidder/proposer is selected. 2) Each proposed block goes through several stages of communication between the nodes before being added and confirmed on the blockchain.

The protocol's implementation relies on a number of components that take turns validating incoming artifacts, proposing new blocks, notarizing validated blocks, and finalizing notarized blocks. This type of loosely coupled implementation makes it easier to ensure that the protocol's security properties are upheld by the implementation. The codebase has reasonable test coverage in terms of unit test cases. However, since a lot of issues that we discovered are also present in *Polygon-edge*, we strongly recommend the DogeChain team to resolve the high severity issues in conjunction with the developers Polygon-edge.

Other recommendations. In accordance with the contract between DogeChain and Veridise, our audit only covered a fraction of the core DogeChain codebase. Going forward, we strongly recommend that the DogeChain team extend their test harness to test for more types of malicious behavior, such as message spam (e.g., nodes spamming the network with duplicate or invalid messages). We also recommend that the DogeChain team implement property tests (e.g., using a framework like quickcheck) to ensure that messages with randomized fields are handled correctly by the artifact validation mechanism. Finally, we also advise the DogeChain team to apply formal methods tools to other critical parts of the code base that were out of scope for the Veridise audit. As formal verification considers all possible edge cases, such technology can uncover deeper logical bugs that are not easy to identify using testing or manual inspection.

Disclaimer. The Veridise model checker takes as input a software artifact and a specification and formally proves that the artifact satisfies the specification in all scenarios (up to some bound). Importantly, the guarantees of the Veridise's model checker are scoped to the provided specification, and the tool does not check any cases not covered by the specification. Furthermore, as standard in software model checking, Veridise also considers all inputs up to some fixed; hence, the guarantees provided by the model checker only pertain to inputs up to that bound.

To complement verification, the Veridise fuzzer utilizes a best-effort strategy to maximize the number of bugs that can be found by the tool within the given time frame. In contrast to the model checker, the fuzzer does not systematically cover all cases; however, it can construct malicious inputs that exceed the bound of the model checker.

We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
DogeChain-consensus	9dc1491	Golang	Native/Linux
DogeChain-contracts	636f666	Solidity	Native/Linux

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
May 1-21, 2022	Manual & Tools	5	15 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Fixed
High-Severity Issues	19	8
Medium-Severity Issues	4	2
Low-Severity Issues	8	4
Informational-Severity Issues	0	1
Undetermined-Severity Issues	0	1

Table 2.4: Category Breakdown.

Name	Number
Auditing & logging	5
Data Validation	20
Logic	6

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of the DogeChain consensus protocol and its smart contract implementation. Specifically, for the DogeChain consensus protocol, we sought to answer the following questions:

- Is the protocol robust to the number of malicious nodes tolerated by IBFT ($\frac{n-1}{3}$)?
- Can a non-validator node ever improperly send a prepare or preprepare message?
- Will an inactive proposer eventually be replaced?
- Can the assignment of a new proposer be controlled by a malicious node?
- Can the validator set be updated after each epoch?
- Can a non-proproser node propose a block?
- Can a node enter the CommitState and insert a block without having received the appropriate number of preprepare/commit messages?
- Will a node in the RoundChange state always eventually exit the RoundChange state?
- Can an ill-formed message ever cause a node to crash?
- Will a validator eventually transition out of the SyncState to the AcceptState?

For DogeChain contracts, we focus on the follow questions:

- Can a stakeholder always reclaim their $DOGE using the Bridge contract?
- Is the reward correctly calculated for both staked $DOGE and staked $DC?
- Can an unstaked user incorrectly withdraw staked funds?
- If a node is made a validator, does it always meet the criteria to become a validator (i.e., have staked at least the minimum threshold and have passed community authority/authentication?
- Can a non-authorized node change the threshold to become a validator?
- Can the number of validators ever fall under the minimum threshold?

3.2 Audit Methodology & Scope

Audit Methodology. Since our goal is to audit a wide spectrum of the DogeChain code base, ranging from low-level consensus protocol implementation to smart contracts at the application level, our audit methodology involved a combination of human experts and wide variety of automated program analysis tools. In particular, we conducted our audit with the aid of the following technologies:

- *Static analysis*. Since the core part of the DogeChain blockchain is written in Golang, we utilized a number of static analysis tools for Golang. In particular, we used static analysis to identify common vulnerabilities in Golang programs as well as suspicious code that is not idiomatic. Such static analysis was performed with the aid of a customized version of the gosec analyzer.

- *Fuzzing*. We also performed fuzz testing to evaluate whether the code behaves correctly under unexpected inputs. Leveraging the `LibAFL` tool, our fuzzing strategy tests the DogeChain consensus protocol by applying *structural mutations* to inputs. Using this fuzzing strategy, we were able to uncover a number of DoS issues as well as logical flaws in the protocol implementation (Section 4).
- *Property-based testing*. We complement fuzzing with another testing technique called *property-based testing*. The goal of this approach is to automatically generate test cases that satisfy a given pre-condition and check that the desired property holds for all generated inputs. We perform property-based testing using a tool called Rapid. In more detail, we formalized important correctness properties of the consensus protocol in Veridise's specification language and implemented a tool to translate these properties to suitable tests in Rapid.
- *Formal verification*. For properties that are not falsified using testing or static analysis, we also used our Veridise infrastructure to perform verification. In particular, we built a bounded verifier for Go on top of the Rosette framework, with the goal of statically checking important correctness properties of the DogeChain consensus protocol. To this end, we formalized key properties of the protocol (Section 5) and used our verifier to statically check whether these properties hold.

Scope. To determine the scope of this audit, we first reviewed the provided documentation on the DogeChain consensus protocol. During this phase, we tried to identify any implicit assumptions made by the protocol, potential edge cases, under-specified components of the protocol, and the behavior of the system contract. Based on the documentation, we formalized key properties to be checked in Veridise's specification language, which are useful both for verification and property-based testing, as described earlier.

In terms of the scope of the audit, the key components we considered include the following:

- The IBFT consensus protocol
- JsonRPC
- System contract for staking
- Helper functions such as keccak, keystore, bridge, etc
- DogeChain application contract

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Moderate
Likely	Warning	Low	Moderate	High
Very Likely	Low	Moderate	High	Critical

In this case, we judge the likelihood of a vulnerability as follows:

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

In addition, we judge the impact of a vulnerability as follows:

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Common Vulnerability Analysis

To evaluate the robustness of DogeChain consensus protocol, part of our audit focused on the detection of common security problems that appear in distributed systems and blockchains. To perform this audit, we first identified a set of common bugs and vulnerabilities from existing databases and built a comprehensive fuzzing infrastructure based on `LibAFL`. Our fuzzer simulates the behavior of a *malicious validator* that actively generates adversarial transactions to exercise edge cases that are unlikely to be revealed by regular unit test cases. Table 4.1 summarizes a list of serious issues that are discovered by our tool so far.

Table 4.1: Summary of Checking for Common Vulnerabilities.

ID	Description	Severity	Status
V-DOGE-VUL-001	Misbehaved node	high	open
V-DOGE-VUL-002	Segmentation fault	high	fixed
V-DOGE-VUL-003	Denial of Service	high	open
V-DOGE-VUL-004	Segmentation fault	high	fixed
V-DOGE-VUL-005	Segmentation fault	high	fixed
V-DOGE-VUL-006	Segmentation fault	high	fixed
V-DOGE-VUL-007	Segmentation fault	high	fixed
V-DOGE-VUL-008	Index out of range	high	fixed
V-DOGE-VUL-009	Panic	high	fixed
V-DOGE-VUL-010	Panic	Moderate	fixed
V-DOGE-VUL-011	divide-by-zero	Low	fixed
V-DOGE-VUL-012	divide-by-zero	Low	fixed
V-DOGE-VUL-013	Test case failed	Low	fixed
V-DOGE-VUL-014	Panic	Low	fixed
V-DOGE-VUL-015	Denial of service	high	open
V-DOGE-VUL-016	Index error in fastrlp	high	Unknown
V-DOGE-VUL-017	Index error in go-web3	Moderate	Unknown

4.1 Detailed Description of Bugs

In this section, we provide a detailed description of each vulnerability.

4.1.1 Misbehaving nodes are not reported or punished by the consensus layer

Severity	High	**Difficulty**	Medium
Type	Data Validation	**Finding ID**	V-DOGE-VUL-001
Target	jury/consensus/ibft.go	**Status**	Open

Description The consensus protocol implemented by DogeChain assumes that at most (N - 1) / 3 nodes are Byzantine, where N is the size of the committee. (To be able to aggregate threshold signature shares for new CUPs, a committee of (2N - 1) / 3 honest nodes is required.) Typically, economic incentives are used to encourage participating nodes to follow the protocol and to punish Byzantine nodes. However, the current implementation of the protocol does not take any action when potentially malicious messages are detected. This is true even for messages that are certain to be malicious, like equivocations from the lowest ranked block maker. This gives Byzantine nodes more freedom to act maliciously without being detected or punished by the honest nodes in the network.

```go
func (i *Ibft) runSyncState() {
    ...
  for i.isState(SyncState) {
    // try to sync with the best-suited peer
    p := i.syncer.BestPeer()
    if p == nil {
      // if we do not have any peers, and we have been a validator
      // we can start now. In case we start on another fork this will be
      // reverted later
      if i.isValidSnapshot() {
        // initialize the round and sequence
        header := i.blockchain.Header()
        i.state.view = &proto.View{
          Round:    0,
          Sequence: header.Number + 1,
        }
        //Set the round metric
        i.metrics.Rounds.Set(float64(i.state.view.Round))

        i.setState(AcceptState)
      } else {
        time.Sleep(1 * time.Second)
      }

      continue
    }
```

Exploit Scenario A malicious node operator executes a denial-of-service attack against a number of nodes in the network. Some of the malicious messages are detected as malformed by honest nodes, but since there is no built-in mechanism to report or punish the attacker, other honest nodes are not alerted, and the malformed messages are simply dropped.

Recommendations In the short term, we recommend improving the validation of incoming messages and revising the validation mechanism so that it returns a different error type for messages that are malicious. We recommend that you ensure that the system broadcasts any malicious behavior across the network so that it becomes known to all nodes.

In the long term, we recommend implementing a slashing mechanism to disincentivize nodes from acting maliciously.

Severity	High	**Difficulty**	Medium
Type	Data Validation	**Finding ID**	V-DOGE-VUL-002
Target	jury/consensus/ibft.go	**Status**	Fixed

4.1.2 Segmentation violation in the consensus module

Description Validators in the set listen on gossips from other validators to form consensus. Developers should assume that these gossips could be from a malicious validator in the network and properly sanitize the incoming messages in the channel. However, the current implementation does not fully validate and sanitize these unsafe messages. Specifically, the proposal attribute of gossiping message could be set to an arbitrary object, including being `nil`, while the following code attempts to dereference this attribute without proper validation. The missing validation could allow a malicious validator to trigger a null pointer dereference on all other validators, thereby crashing all of them or leading to denial of service (or potentially even more serious consequences).

```go
func (i *Ibft) runAcceptState() {
    ...
  for i.getState() == AcceptState {
    msg, ok := i.getNextMessage(timeout)
    if !ok {
      return
    }

    if msg == nil {
      i.setState(RoundChangeState)
      continue
    }

    if msg.From != i.state.proposer.String() {
      i.logger.Error("msg received from wrong proposer")
      continue
    }

    // retrieve the block proposal
    block := &types.Block{}
    if err := block.UnmarshalRLP(msg.Proposal.Value); err != nil {
      i.logger.Error("failed to unmarshal block", "err", err)
      i.setState(RoundChangeState)

      return
    }
  ...
  }
}
```

Exploit Scenario An attacker owning a validator node could send a maliciously-crafted gossip message to all validators and conduct a denial of service attack to bring down all validators in the network. With no validator in the network, no new block would be formed and no new transaction could be processed. An attacker could also make their validator node the only validator to survive in the validators set and conduct 51% attack, allowing them to manipulate transactions in the new blocks.

Recommendations Consider adding a validation check as following:

```
func (i *Ibft) runAcceptState() {
    ...
  for i.getState() == AcceptState {
    msg, ok := i.getNextMessage(timeout)
    if !ok {
      return
    }

    if msg == nil {
      i.setState(RoundChangeState)
      continue
    }

    if msg.From != i.state.proposer.String() {
      i.logger.Error("msg received from wrong proposer")
      continue
    }

+   if msg.Proposal == nil {
+     // A malicious node conducted a DoS attack
+     continue
+   }

    // retrieve the block proposal
    block := &types.Block{}
    if err := block.UnmarshalRLP(msg.Proposal.Value); err != nil {
      i.logger.Error("failed to unmarshal block", "err", err)
      i.setState(RoundChangeState)

      return
    }
  ...
  }
}
```

4.1.3 Denial of service in the transaction pool module

Severity	High	**Difficulty**	Medium
Type	Data Validation	**Finding ID**	V-DOGE-VUL-003
Target	jury/consensus/ibft.go	**Status**	Open

Description The transaction pool module sets a threshold on amount of transactions that could be kept inside the transaction pool (i.e., having more transactions than threshold makes the transaction pool `overflow`). The threshold is tracked by using counter inside `slotGauge` interface. When the counter is equal to the threshold, no new transactions would be added to the transaction pool. By flooding the JSON RPC interface with significant amount of valid transactions (e.g., 40960 transactions when threshold is 4096), it is possible to make the counter of `slotGauge` equal to the threshold while never get decremented, even when the transaction pool is empty later. This bug could make the transaction pool locked and throw `ErrTxPoolOverflow` exceptions for all following new transactions. We are still investigating the reason why the counter is not decremented after transactions are removed from the transaction pool.

```go
func (p *TxPool) addTx(origin txOrigin, tx *types.Transaction) error {
    ...
  // check for overflow
  if p.gauge.read()+slotsRequired(tx) > p.gauge.max {
    return ErrTxPoolOverflow
  }
  ...
}
```

Exploit Scenario A malicious client with funds (for gas fee of the transactions) can conduct denial of service. Such an attack could cause all nodes in the network no longer able to accept new transactions permanently unless they are restarted or reset.

Recommendations In the short term, we recommend removing the overflow check in `addTx` function because the transaction pool can support as many transactions as memory of the node can tolerate. Another option could be setting `sloGauge.max` to be a very large value.

In the long term, we recommend investigating the true root cause of the bug and fix it in a more principled way.

4.1.4 Segmentation violation during block gossiping A1

Severity	High	**Difficulty**	Easy
Type	Data Validation	**Finding ID**	V-DOGE-VUL-004
Target	jury/protocol/service_v1.go	**Status**	Fixed

Description All participating nodes in the blockchain listen on the gossip of new blocks leveraging Protobuf and libp2p. Developers should not assume that these gossips are from valid nodes and should properly validate and sanitize the gossip messages. Specifically, the proposal attribute of gossiping message could be set to an arbitrary object, including `nil`, but the following code attempts to dereference this attribute without proper validation. The missing validation could allow a malicious validator to trigger a null pointer dereference on all other nodes (including validators), thus crashing all of them and leading to denial of service or potentially more serious consequences.

```go
func (s *serviceV1) Notify(ctx context.Context, req *proto.NotifyReq) (*empty.Empty, error) {
    ...

  b := new(types.Block)
  if err := b.UnmarshalRLP(req.Raw.Value);
  ...
}
```

Exploit Scenario Although the endpoint is intended to listen only on messages from validators (consensus module), anyone can access such an endpoint. As a result, any attacker could send a maliciously-crafted gossip message to all nodes, including validators, and conduct a denial of service attack to bring down all nodes in the network. With no nodes in the network, no new block would be formed and no new transactions could be processed. An attacker could also make their validator node the only validator to survive in the validators set and conduct a 51% attack, allowing them to manipulate transactions in the new blocks.

Recommendations Consider adding a validation check as following:

```
func (s *serviceV1) Notify(ctx context.Context, req *proto.NotifyReq) (*empty.Empty, error) {
    ...

  b := new(types.Block)
+ if req.Raw == nil {
+     // malicious node conducted denial of service
+     return &empty.Empty{}, nil
+ }
  if err := b.UnmarshalRLP(req.Raw.Value);
  ...
}
```

4.1.5 Segmentation violation during block gossiping B1

Severity	High	**Difficulty**	Medium
Type	Data Validation	**Finding ID**	V-DOGE-VUL-005
Target	jury/protocol/service_v1.go	**Status**	Fixed

Description All participating nodes in the blockchain request information from peers leveraging Protobuf and libp2p. Developers should not assume that the peers of a node are benign and should properly validate or sanitize their responses. Specifically, the block syncer's requests header from peers and parse the responses. The `Spec` attribute of responses could be set to an arbitrary object, including being `nil`, while the following code attempts to dereference this attribute without proper validation. The missing validation could allow a malicious node to trigger a null pointer dereferencing on all its peers (including validators), thereby crashing them or leading to denial of service (or potentially more serious consequences).

```
func getHeaders(clt proto.V1Client, req *proto.GetHeadersRequest) ([]*types.Header, error) {
  resp, err := clt.GetHeaders(context.Background(), req)
    ...
  for _, obj := range resp.Objs {
    header := &types.Header{}
    if err := header.UnmarshalRLP(obj.Spec.Value); err != nil {
      return nil, err
    }

    headers = append(headers, header)
  }

  return headers, nil
}
```

Exploit Scenario A malicious node could send a maliciously-crafted response to all its peers, including validators, and conduct a denial of service attack to bring down nodes in the network.

A resourceful attacker could also leverage this vulnerability to bring down all nodes in the network, as long as they trick all nodes to add their malicious node as a peer. With no nodes in the network, no new block would be formed and no new transaction could be processed. An attacker could also make their validator node the only validator to survive in the validators set and conduct a 51% attack, allowing them to manipulate transactions in the new blocks.

Recommendations Consider adding a validation check as following:

```
func getHeaders(clt proto.V1Client, req *proto.GetHeadersRequest) ([]*types.Header, error) {
  resp, err := clt.GetHeaders(context.Background(), req)
    ...
  for _, obj := range resp.Objs {
    header := &types.Header{}
+   if obj.Spec == nil { continue }
    if err := header.UnmarshalRLP(obj.Spec.Value); err != nil {
      return nil, err
    }

    headers = append(headers, header)
  }

  return headers, nil
}
```

4.1.6 Segmentation fault during block gossiping B2

Severity	High	**Difficulty**	Medium
Type	Data Validation	**Finding ID**	V-DOGE-VUL-006
Target	jury/protocol/syncer.go	**Status**	Fixed

Description All participating nodes in the blockchain request information from peers leveraging Protobuf and libp2p. Developers should not assume that the peers of a node are benign and should properly validate or sanitize their responses. Specifically, the block syncer's requests header from peers and parse the responses. The `Spec` attribute of responses could be set to an arbitrary object, including `nil`, but the following code attempts to dereference this attribute without proper validation. The missing validation could allow a malicious node to trigger a null pointer dereference on all its peers (including validators), thereby causing them to crash or resulting in denial of service (or potentially even more serious consequences).

```
func getHeader(clt proto.V1Client, num *uint64, hash *types.Hash) (*types.Header, error) {
  req := &proto.GetHeadersRequest{}
  ...

  resp, err := clt.GetHeaders(context.Background(), req)
  if err != nil {
    return nil, err
  }

  if len(resp.Objs) == 0 {
    return nil, nil
  }

  if len(resp.Objs) != 1 {
    return nil, fmt.Errorf("unexpected more than 1 result")
  }

  header := &types.Header{}

  if err := header.UnmarshalRLP(resp.Objs[0].Spec.Value); err != nil {
    return nil, err
  }

  return header, nil
}
```

Exploit Scenario A malicious node could respond a maliciously-crafted response to all its peers, including validators, and conduct a denial of service attack to bring down nodes in the network. A resourceful attacker could also leverage this vulnerabilities to bring down all nodes in the network, as long as they trick all nodes to add their malicious node as a peer. With no nodes in the network, no new block would be formed and no new transaction could be processed. An attacker could also make their validator node the only validator to survive in the validators set and conduct 51% attack, allowing them to manipulate transactions in the new blocks.

Recommendations Consider adding a validation check as following:

```
func getHeader(clt proto.V1Client, num *uint64, hash *types.Hash) (*types.Header, error) {
  req := &proto.GetHeadersRequest{}
  ...

  resp, err := clt.GetHeaders(context.Background(), req)
  if err != nil {
    return nil, err
  }

  if len(resp.Objs) == 0 {
    return nil, nil
  }

  if len(resp.Objs) != 1 {
    return nil, fmt.Errorf("unexpected more than 1 result")
  }

  header := &types.Header{}

+ if resp.Objs[0].Spec == nil {
+   return nil, err
+ }

  if err := header.UnmarshalRLP(resp.Objs[0].Spec.Value); err != nil {
    return nil, err
  }

  return header, nil
}
```

4.1.7 Segmentation fault during transaction gossiping

Severity	High	**Difficulty**	Easy
Type	Data Validation	**Finding ID**	V-DOGE-VUL-007
Target	jury/txpool/txpool.go	**Status**	Fixed

Description Validators listen on gossips from other validators to form consensus. Developers should assume that these gossips could be from a malicious validator in the network and properly sanitize the incoming messages in the channel. However, the current implementation does not fully validate and sanitize these unsafe messages. Specifically, the `raw` attribute of gossiping message could be set to an arbitrary object, including `nil`, but the following code attempts to dereference this attribute without proper validation. The missing validation could allow a malicious node to trigger a null pointer dereference on all validators, thus crashing all of them, leading to denial of service or potentially more serious consequences.

```go
func (p *TxPool) addGossipTx(obj interface{}) {
  if !p.sealing {
    return
  }

  raw := obj.(*proto.Txn) // nolint:forcetypeassert
  tx := new(types.Transaction)

  // decode tx
  if err := tx.UnmarshalRLP(raw.Raw.Value); err != nil {
    p.logger.Error("failed to decode broadcasted tx", "err", err)

    return
  }

  // add tx
  if err := p.addTx(gossip, tx); err != nil {
    p.logger.Error("failed to add broadcasted txn", "err", err)
  }
}
```

Exploit Scenario An attacker could send a maliciously-crafted gossip message to all validators and conduct a denial of service attack to bring down all validators in the network. With no validator in the network, no new block would be formed and no new transaction could be processed. An attacker could also make their validator node the only validator to survive in the validators set and conduct 51% attack, allowing them to manipulate transactions in the new blocks.

Recommendations Consider adding a validation check like the following:

```go
func (p *TxPool) addGossipTx(obj interface{}) {
  if !p.sealing {
    return
  }

  raw := obj.(*proto.Txn) // nolint:forcetypeassert
  tx := new(types.Transaction)

+   if raw.Raw == nil { return }

  // decode tx
  if err := tx.UnmarshalRLP(raw.Raw.Value); err != nil {
    p.logger.Error("failed to decode broadcasted tx", "err", err)

    return
  }

  // add tx
  if err := p.addTx(gossip, tx); err != nil {
    p.logger.Error("failed to add broadcasted txn", "err", err)
  }
}
```

4.1.8 Index out of range in consensus message verification

Severity	High	**Difficulty**	Easy
Type	Data Validation	**Finding ID**	V-DOGE-VUL-008
Target	jury/consensus/ibft.go	**Status**	Fixed

Description Validators listen on gossips from other validators to form consensus. Developers should assume that these gossips could be from a malicious validator in the network and properly sanitize the incoming messages in the channel. The current implementation uses `ecrecoverImpl` to verify the signature of the gossip messages from other validators, which calls `RecoverPubkey`. If the signature is crafted to be an empty string, the following code would panic with an index out of range exception. The missing validation could allow a malicious node to trigger a panic on all validators, thus crashing all of them, leading to denial of service or potentially more serious consequences.

```go
func RecoverPubkey(signature, hash []byte) (*ecdsa.PublicKey, error) {
  size := len(signature)
  term := byte(27)

  if signature[size-1] == 1 {
    term = 28
  }

  sig := append([]byte{term}, signature[:size-1]...)
  pub, _, err := btcec.RecoverCompact(S256, sig, hash)

  if err != nil {
    return nil, err
  }

  return pub.ToECDSA(), nil
}
```

Exploit Scenario An attacker could send a maliciously-crafted gossip message to all validators and conduct a denial of service attack to bring down all validators in the network. With no validator in the network, no new block would be formed and no new transaction could be processed. An attacker could also make their validator node the only validator to survive in the validators set and conduct 51% attack, allowing them to manipulate transactions in the new blocks.

Recommendations Consider adding a validation check as following:

```
func RecoverPubkey(signature, hash []byte) (*ecdsa.PublicKey, error) {
  size := len(signature)
+ if size == 0 {
+     return nil, fmt.Errorf("empty signature")
+ }
  term := byte(27)

  if signature[size-1] == 1 {
    term = 28
  }

  sig := append([]byte{term}, signature[:size-1]...)
  pub, _, err := btcec.RecoverCompact(S256, sig, hash)

  if err != nil {
    return nil, err
  }

  return pub.ToECDSA(), nil
}
```

4.1.9 Panic in the consensus module

Severity	High	**Difficulty**	Medium
Type	Data Validation	**Finding ID**	V-DOGE-VUL-009
Target	jury/consensus/ibft.go	**Status**	Fixed

Description Validators in the set listen on gossips from other validators to form consensus. Developers should consider that these gossips could be from a malicious validator in the network and properly sanitize the incoming messages in the channel. However, the current implementation has not fully validate and sanitize these unsafe messages. Specifically, the proposal attribute of gossiping message could be set to an arbitrary value, while the following code can only parse hex value and panic otherwise. The missing validation could allow a malicious validator to trigger a panic on all other validators, thus crashing all of them, leading to denial of service or potentially more serious consequences.

```
func (i *Ibft) insertBlock(block *types.Block) error {
  committedSeals := [][]byte{}
  for _, commit := range i.state.committed {
    // no need to check the format of seal here because writeCommittedSeals will check
    committedSeals = append(committedSeals, hex.MustDecodeHex(commit.Seal))
  }
  ...
}
```

Exploit Scenario An attacker could send a maliciously-crafted gossip message to all nodes, including validators, and conduct a denial of service attack to bring down all nodes in the network. With no nodes in the network, no new block would be formed and no new transaction could be processed. An attacker could also make their validator node the only validator to survive in the validators set and conduct 51% attack, allowing them to manipulate transactions in the new blocks.

Recommendations Consider adding a validation check, removing the panic, or adding a mechanism for recovering from panic.

4.1.10 Panic in send transaction JSON RPC endpoint

Severity	Moderate	**Difficulty**	Easy
Type	Data Validation	**Finding ID**	V-DOGE-VUL-010
Target	jury/jsonrpc/eth_endpoint.go	**Status**	Fixed

Description All nodes have JSON RPC endpoints and they are intended to be exposed to the public. Developers should consider that the RPC requests could be from a malicious actor and properly validate and sanitize the request before conducting further processing. Specifically, in send transaction JSON RPC endpoint, the code attempts to parse the transaction string as a hex value, and panics otherwise. Although such a panic is later recovered, it may cause potential issues including resource leak in future development.

```go
func (e *Eth) SendRawTransaction(input string) (interface{}, error) {
  buf := hex.MustDecodeHex(input)
    ...
}

```

Exploit Scenario N/A

Recommendations Consider adding a validation check, removing the panic, or adding a mechanism for recovering from panic.

4.1.11 Potential divided-by-zero in `GetEpoch`

Severity	Low	**Difficulty**	Easy
Type	Data Validation	**Finding ID**	V-DOGE-VUL-011
Target	jury/consensus/ibft.go	**Status**	Fixed

Description In `ibft.go`, line 1297: the function `getEpoch` performs a check of the following: `number%i.epochSize == 0`. If `i.epochSize` is equal to 0, this will crash. Right now, the only way that `epochSize` can be updated is from the command line when the protocol is started, so this is not a high-priority issue. However, if the code changes in the future to allow the epoch size to be updated, this could cause a crash.

```go
// GetEpoch returns the current epoch
func (i *Ibft) GetEpoch(number uint64) uint64 {
    if number%i.epochSize == 0 {
        return number / i.epochSize
    }
    return number/i.epochSize + 1
}
```

Exploit Scenario N/A

Recommendations To fix it, just add a check for epoch size of 0 to the function.

4.1.12 Potential division-by-zero in `IsLastOfEpoch`

Severity	Low	**Difficulty**	Easy
Type	Data Validation	**Finding ID**	V-DOGE-VUL-012
Target	jury/consensus/ibft.go	**Status**	Fixed

Description In `ibft.go`, line 1307: the function `IsLastOfEpoch` performs a check of the following: `number%i.epochSize == 0`. If `i.epochSize` is equal to 0, this will crash. Right now, the only way that `epochSize` can be updated is from the command line when the protocol is started, so this is not a high-priority issue. However, if the code changes in the future to allow the epoch size to be updated, this could cause a crash.

```go
// IsLastOfEpoch checks if the block number is the last of the epoch
func (i *Ibft) IsLastOfEpoch(number uint64) bool {
      return number > 0 && number%i.epochSize == 0
}
```

Exploit Scenario N/A

Recommendations To fix it, just add a check for epoch size of 0 to the function.

4.1.13 `TestExtraEncoding` in extra_test.go fails on edge cases

Severity	Low	**Difficulty**	Easy
Type	Test Suite	**Finding ID**	V-DOGE-VUL-013
Target	jury/consensus/extra_test.go	**Status**	Fixed

Description This issue only arises when we instantiate an `IstanbulExtra` struct and pass empty values for its fields. The JSON encoding/decoding process seems to be correct (We've printed the values and they are the same), but the `reflect.DeepEqual` fails. We suppose it is a problem with the reflect package, not with the actual code.

```
&IstanbulExtra{
    Validators: []types.Address{},
    Seal: []byte{},
  CommittedSeal: [][]byte{},
}
```

Exploit Scenario N/A

Recommendations To fix it, just add a check for empty struct.

4.1.14 Potential panic in handling message

Severity	Low	**Difficulty**	Easy
Type	Data Validation	**Finding ID**	V-DOGE-VUL-014
Target	jury/consensus/ibft.go	**Status**	Fixed

Description Here, adding a new message type could realistically trigger this panic. Instead, a better way to prevent a bug of this nature would be to (1) enable error handling that logs the issue rather than panicing and (2) writing unit tests that ensure the issue does not happen.

```
func (i *Ibft) runValidateState() {
  hasCommitted := false
  sendCommit := func() {
    ...
  switch msg.Type {
    case proto.MessageReq_Prepare:
      i.state.addPrepared(msg)

    case proto.MessageReq_Commit:
      i.state.addCommitted(msg)

    default:
      panic(fmt.Sprintf("BUG: %s", reflect.TypeOf(msg.Type)))
  }
```

Exploit Scenario N/A

Recommendations We would recommend writing a test that sends a message of every type while a particular validator is in the ValidateState and ensure that the validator only processes commit and prepare messages.

4.1.15 Denial of service in RoundChangeState

Severity	High	**Difficulty**	Medium
Type	Data Validation	**Finding ID**	V-DOGE-VUL-015
Target	jury/consensus/msg_queue.go	**Status**	Open

Description A node in the `RoundChange` state uses the `readMessage` method to read messages from other nodes to reach a consensus on the next round. Upon receiving, these messages are stored in a queue and sorted by the view (i.e., the position inside the blockchain) of the sender node, firstly by sequence and then by round. While having the messages sorted is the intended behavior since it allows the `readMessage` method to discard old/future messages easily, it can also be a source of vulnerability, as presented below.

```go
func (m msgQueueImpl) Less(i, j int) bool {
  ti, tj := m[i], m[j]
  // sort by sequence
  if ti.view.Sequence != tj.view.Sequence {
    return ti.view.Sequence < tj.view.Sequence
  }
  // sort by round
  if ti.view.Round != tj.view.Round {
    return ti.view.Round < tj.view.Round
  }
  // sort by message
  return ti.msg < tj.msg
}
```

Exploit Scenario A malicious node could perform a Denial of Service attack by spamming nodes in the `RoundChange` state with crafted messages that have sequences lower than the sequences of the attacked nodes. Consequently, the attacker's messages will be added to the top of the queue. This attack makes the nodes in the `RoundChange` state considerably slow, as they have to read the attacker's messages first. Based on our experiments, this attack could lead to a block generation time delayed by 350 seconds instead of 2 seconds, which is the normally expected delay.

Recommendations We encourage the developers to find an alternative for storing/reading messages that mitigates the risks presented above.

4.1.16 Index out of range in fastrlp

Severity	High	**Difficulty**	High
Type	Data Validation	**Finding ID**	V-DOGE-VUL-016
Target	umbracle/fastrlp/parser.go	**Status**	Open

Description The blockchain implementation heavily leverages third-party dependency `fastrlp` for serialization and deserialization. Specifically, endpoints leverage `fastrlp.Parser.Parse` to parse user inputs in bytes like transactions and accounts. However, this function does not properly validate the input and directly conducts deserialization. Passing unsanitized messages can cause an index out of range panic, leading to denial of service and bringing down the node.

In the code below, the `parseValue` function attempts to use the first byte of the byte array (`b`) provided by users to determine its length. If the first byte is `0xff` (i.e., b[0]= 0xf9), this function would attempt to read two bytes from the array. If the byte array length is only one, then an index out of range panic is thrown.

```go
func parseValue(b []byte, c *cache) (*Value, []byte, error) {
  if len(b) == 0 {
    return nil, b, fmt.Errorf("cannot parse empty string")
  }

  cur := b[0]
  ...

  intSize := int(cur - 0xF7)
  size := readUint(b[1:intSize+1], c.buf[:])
  if size < 56 {
    return nil, nil, fmt.Errorf("bad size")
  }
  v, tail, err := parseList(b[intSize+1:], intSize, int(size), c)
  if err != nil {
    return nil, tail, fmt.Errorf("cannot parse long array: %s", err)
  }
  return v, tail, nil
}
```

Function `fastrlp.Parser.Parse` provided by the third-party dependency is widely used throughout the code. One example is deserializing transactions in bytes supplied by the users or validators in libp2p endpoint and gRPC endpoint, which all leverages the function below.

```go
func UnmarshalRlp(obj unmarshalRLPFunc, input []byte) error {
  pr := fastrlp.DefaultParserPool.Get()

  v, err := pr.Parse(input)
  if err != nil {
    fastrlp.DefaultParserPool.Put(pr)

    return err
  }

  if err := obj(pr, v); err != nil {
    fastrlp.DefaultParserPool.Put(pr)

    return err
  }

  fastrlp.DefaultParserPool.Put(pr)

  return nil
}
```

Exploit Scenario There can be multiple ways to conduct a DoS attack to exploit this vulnerability in `fastrlp`. The easiest one could be an attacker sending a maliciously-crafted gossip message to all nodes, conducting a denial of service attack to bring down all nodes in the network. With no validator in the network, no new block would be formed and no new transaction could be processed. An attacker could also make their validator node the only validator to survive in the validators set and conduct 51% attack, allowing them to manipulate transactions in the new blocks.

Recommendations Consider adding a bound checking before reading elements of slices or adding a recover after using the vulnerable function.

4.1.17 Index out of range in go-web3 (ethgo)

Severity	Moderate	**Difficulty**	High
Type	Data Validation	**Finding ID**	V-DOGE-VUL-017
Target	umbracle/ethgo/abi/abi.go	**Status**	Open

Description The transaction event handler leverages third-party dependency `go-web3` (`ethgo`) for parsing event logs. Specifically, blockchain leverages `ethgo.Event.ParseLog` to parse bridge event logs from bridge contract. However, this function does not properly validate the input and directly conducts parsing. Passing unsanitized messages can cause an index out of range panic, leading to denial of service and bringing down the node.

In the code below, the `decode` function used by `ParseLog` function to parse data attempts to read `32+length` bytes from user input but there is only a validation check of whether user input length is greater than or equal to 32.

```go
func decode(t *Type, input []byte) (interface{}, []byte, error) {
  var data []byte
  var length int
  var err error

  // safe check, input should be at least 32 bytes
  if len(input) < 32 {
    return nil, nil, fmt.Errorf("incorrect length")
  }

  if t.isVariableInput() {
    length, err = readLength(input)
    if err != nil {
      return nil, nil, err
    }
  } else {
    data = input[:32]
  }

  switch t.kind {
  ...
  switch t.kind {
  case KindBool:
    val, err = decodeBool(data)

  case KindInt, KindUInt:
    val = readInteger(t, data)

  case KindString:
    val = string(input[32 : 32+length])

  case KindBytes:
    val = input[32 : 32+length]
    ...
    }
    ...
}
```

Exploit Scenario A resourceful attacker can potentially leverage this vulnerability to conduct DoS attack by chaining with other bugs in consensus module.

Recommendations Consider adding a bounds check before reading elements of slices or adding a recover after using the vulnerable function.

5 Formalization and Analysis of Custom Correctness Properties

5.1 Summary of Custom Properties

The expected behavior of the IBFT protocol can be summarized using the state machine in Figure 5.1. The state machine defines a set of crucial *states* and *transitions*.

The original IBFT paper defines four states:

- *Awaiting Proposal*. Validator is waiting for a new block to be supplied by the current proposer. If the validator is the proposer for this round, they prepare the proposed block and transmit it in a pre-prepare message.
- *Preparing*. Has received a (valid) proposed block and notified validator-peers; is now waiting for validator-peers to notify their acceptance of the block.
- *Ready*. Has received validator-peer's acceptance of block, and is waiting for them to be a in a similar position. At this stage the proposed block has been 'locked-in', and cannot be replaced until an attempt at insertion has been conducted.
- *Round Change*. The round timed out before consensus was reached or the block failed to insert. Wait for all validators to agree on the next round number.

There are six crucial transitions:

- *Awaiting Proposal* −→ *Preparing*. On reception of a new block (Preprepare message) from the proposer (i.e. the block is valid in its content, as is its proposed chain insertion point).
- *Awaiting Proposal* −→ *Round Change*. The received proposal was not a valid block according to a given set of rules (e.g. invalid proposer, incorrect round numbering).
- *Preparing* −→ *Ready*. On reception of 2F+1 notifications (Prepare message) from validator-peers indicating the proposed block is suitable for insertion.
- *Ready* −→ *Awaiting Proposal*. On reception of 2F+1 notifications (Commit message) from validator-peers indicating they are ready to append the block to the chain. On transition, the process of appending the block to the chain is performed (success).
- *Ready* −→ *Round Change*. As per Ready->Awaiting Proposal, however, block insertion has failed.
- *Round Change* −→ *Awaiting Proposal*. 2F+1 of validators agree on the next round number to be used.

Table 5.1: Summary of Formal Verification.

ID	Description	Severity	Status
V-DOGE-FV-001	RoundChange state	high	open
V-DOGE-FV-002	Failing transition	high	open
V-DOGE-FV-003	Incorrect timeout	high	open
V-DOGE-FV-004	Incorrect timeout	high	open
V-DOGE-FV-005	Incorrect timeout	high	open
V-DOGE-FV-006	Failing transition	high	open

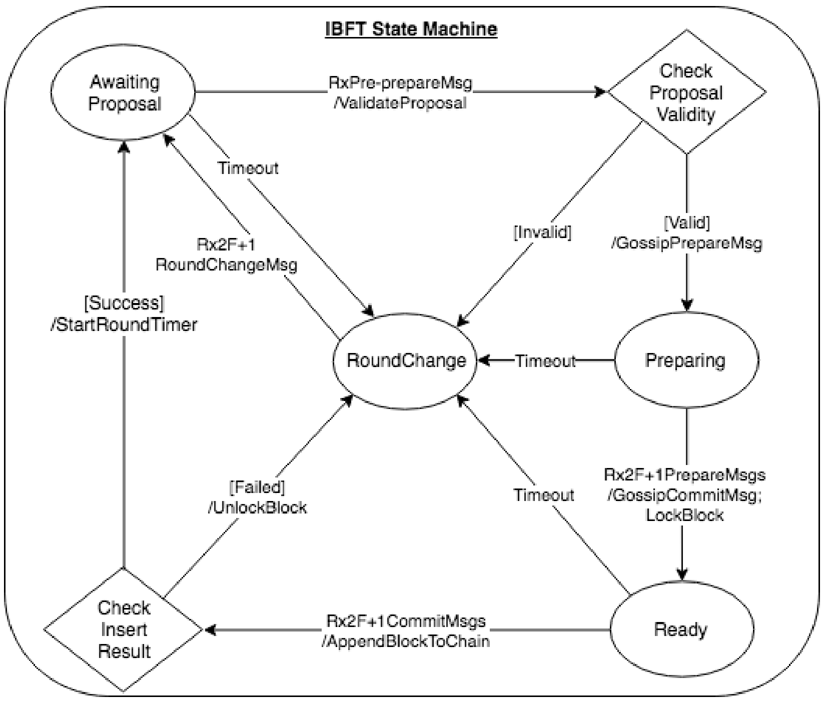

Figure 5.1: IBFT State Machine.

5.2 Formalization

Specification Syntax To allow auditors and developers to specify properties, we extend Golang and Solidity's syntax with temporal quantifiers. In particular, we make use of the following operators from Linear Temporal Logic (LTL):

- `always`, denoted $\square$. $\square\phi$ indicates the condition ϕ (e.g. $owner = 0x0$) is true at all times including and after the current point during execution of the program.
- `eventually`, denoted $\diamond$. $\diamond\phi$ indicates that either ϕ holds at the current point or ϕ will eventually hold at some point in the future.

For instance, $\square(\text{claimRefund}() \rightarrow \text{sum}(\text{deposits}) < 10000)$ represents that "Investors cannot claim refunds while more than 10,000 ether is collected."

V-DOGE-FV-001: A node in the RoundChange state will always eventually exit the RoundChange state:

Variables	Ibft i
Property	$\square(i.getState() = \text{RoundChange} \rightarrow \diamond i.getState() \neq \text{RoundChange})$

V-DOGE-FV-002: A node in AwaitingProposal will always eventually proceed to the Preparing state after receiving a valid preprepare message from the proposer:

Variables	Ibft i, Message m
Property	$\square((i.getState() = \textsc{AwaitingProposal} \land \text{received}(i, m) \land \text{isValid}(m) \land \text{isPreprepare}(m) \land \text{isProposer}(m.sender)) \rightarrow \diamond i.getState() = \textsc{Preparing})$

V-DOGE-FV-004: A node in the AwaitingProposal state will always eventually exit the AwaitingProposal state:

Variables	Ibft i
Property	$\square(i.getState() = \textsc{AwaitingProposal} \rightarrow \diamond i.getState() \neq \textsc{AwaitingProposal})$

V-DOGE-FV-006: A node in the Preparing state will always eventually exit the Preparing state:

Variables	Ibft i
Property	$\square(i.getState() = \textsc{Preparing} \rightarrow \diamond i.getState() \neq \textsc{Preparing})$

V-DOGE-FV-009: A node in the Ready state will always eventually exit the Ready state:

Variables	Ibft i
Property	$\square(i.getState() = \textsc{Ready} \rightarrow \diamond i.getState() \neq \textsc{Ready})$

V-DOGE-FV-010: A node in Preparing will always proceed to the Ready state after it has received at least $2F + 1$ prepare messages:

Variables	Ibft i
Property	$\square(i.getState() = \textsc{Preparing} \land \text{numPrepareReceived}(i) > 2F) \rightarrow \diamond(i.getState() = \textsc{Ready})$

5.3 Results

In this section, we provide a detailed description of properties that are violated by the implementation.

5.3.1 Picking a new round indefinitely

Severity	High	**Difficulty**	Medium
Type	Data Validation	**Finding ID**	V-DOGE-FV-001
Target	jury/consensus/ibft.go	**Status**	Open

Description Since there is no bound on the number of times validator nodes can propose new rounds in the RoundChangeState, it is possible to get into a state where all validator nodes remain in the RoundChangeState indefinitely. This occurs when all validator nodes continuously propose different round numbers. Since a consensus is never reached, the locally proposed round numbers of each validator node increase separately, resulting in the nodes to never agree.

There are two ways that the consensus protocol could exit this loop, both of which may not happen in some executions. The first way that the consenus protocol could exit this state is by some node entering the SyncState. However, since this will only happen when some validator node's blockchain is out of sync, breaking out of this bad state would not be possible when all validator nodes' blockchain is in sync. The second way the system could escape all nodes being stuck in the RoundChangeState is by waiting for enough validators to propose the same round number. If an execution occurs where all validator nodes execute in lock-step, this will never happen, and all validator nodes will be trapped in RoundChangeState.

The code below shows the execution loop for RoundChangeState. The function checkTimeout can move the validator to the state SyncState and the num == i.state.NumValid() branch can move the validator to AcceptState, but these cases can both be avoided as discussed above.

```go
func (i *Ibft) runRoundChangeState() {
    checkTimeout := func() {
    ... // if there is an advanced peer, do i.setState(SyncState)

    // otherwise, it seems that we are in sync
    // and we should start a new round
    sendNextRoundChange()
  }
    ...
  for i.getState() == RoundChangeState {
    msg, ok := i.getNextMessage(timeout)
    if !ok {
      // closing
      return
    }

    if msg == nil {
      i.logger.Debug("round change timeout")
      checkTimeout()
      // update the timeout duration
      timeout = exponentialTimeout(i.state.view.Round)

      continue
    }

    // we only expect RoundChange messages right now
    num := i.state.AddRoundMessage(msg)

    if num == i.state.NumValid() {
      // start a new round immediately
      i.state.view.Round = msg.View.Round
      i.setState(AcceptState)
    } else if num == i.state.validators.MaxFaultyNodes()+1 {
      // weak certificate, try to catch up if our round number is smaller
      if i.state.view.Round < msg.View.Round {
            // update timer
            timeout = exponentialTimeout(i.state.view.Round)
            sendRoundChange(msg.View.Round)
          }
        }
      }
      ...
    }
```

Exploit Scenario This vulnerability can be triggered in two ways, one through malicious execution and one through standard execution. To maliciously trigger this vulnerability, a single malicious validator node sends enough round change messages to each other altruistic validator that all altruistic validators are in different rounds. From this state, if all altruistic validators execute in lock step (or near lock step), the validators will never agree on a round number and will remain in RoundChangeState indefinitely. A similar behavior can occur without a malicious validator simply if some round change messages from an altruistic validator get dropped. Specifically, if enough round change messages are dropped going to each validator such that the system appears to be in a different round to each validator, then the system could reach this state.

Recommendation In the short term, validators can propose round number 0 when the number of iterations in the RoundChangeState loop surpasses some maximum threshold.

```
func (i *Ibft) runRoundChangeState() {
    ...
  for i.getState() == RoundChangeState {
      // check for
      if iterations > MAX_ITERATIONS {
          sendRoundChange(0)
          // Do not perform checkTimeout() in this case
      }
      ...
  }
  ...
}
```

In the long term, a proof that non-compromised validators will always eventually exit the round change state would ensure that bugs like these do not appear again.

5.3.2 Failing AwaitingProposal-to-Preparing transition

Severity	High	**Difficulty**	Medium
Type	Data Validation	**Finding ID**	V-DOGE-FV-002
Target	jury/consensus/ibft.go	**Status**	Open

Description Based on the implementation of `getNextMessage(...)`, validators must process all messages of the correct message type. Since duplicate messages are allowed, a malicious validator can send an arbitrary number of messages to delay the processing of a valid proposal. In `runAcceptState()`:

```go
func (i *Ibft) runAcceptState() {
    ...
    for i.getState() == AcceptState {
    msg, ok := i.getNextMessage(timeout)

        ... // Move to RoundChangeState if msg == nil

    if msg.From != i.state.proposer.String() {
      i.logger.Error("msg received from wrong proposer")

      continue
    }
    ...
}
```

As discussed in the previous section, this `if` statement allows a malicious node to force the validator to continually process messages. However, depending on messages received from other non-malicious nodes, the correct behavior here may not be a timeout. Specifically, consider the case where the validator has received a proposal from a non-malicous proposer. Based on the current implementation, the non-malicoius validator that receives the proposal must first process an arbitrary number of messages from the malicious validator before it can process the message from the proposer.

Ideally, the validator will spend a bounded amount of time processing malicoius messages so that after receiving a valid proposal, it is still able to transition to the ValidateState (given a reasonable timeout).

Exploit Scenario To trigger the bug, consider the case where there is exactly one malicious node, and all nodes execute normally until reaching the AcceptState. At this point, the malicious node (who is not the proposer) should repeatedly send prepare messages. Each non-malicious node will execute `i.state.addPrepared(msg)` upon receiving the message, but since `i.state.numPrepared()` counts the number of prepare messages sent from *distinct* validators, `sendCommit()` will never execute. Similarly, since the malicious node is repeatedly sending new messages, all other validators will never timeout. Thus, a denial-of-service will be achieved.

Though this is similar to the exploit in subsection 5.3.3, the cause is slightly different. As a result, the fix for this vulnerability requires more than adding a timeout to `runAcceptState()`.

Recommendation In the short term, processing only one message from each validator per state transition (or equivalently, dropping duplicate received messages) would eliminate this exploit. It would also mean that arbitrary of malicious validator messages, nodes in the AcceptState will correctly transition whenever they receive a proposal that can be validated (given a reasonable timeout).

In the long term, additional test cases should be written to show that whenever a validatable proposal is received in the AcceptState from the proposer, the validator transitions to the ValidateState.

5.3.3 Incorrect timeouts

Severity	High	**Difficulty**	Medium
Type	Data Validation	**Finding ID**	V-DOGE-FV-003 V-DOGE-FV-004 V-DOGE-FV-005
Target	jury/consensus/ibft.go	**Status**	Open

Description In the `runCycle()` execution loop, timeouts only occur whenever no message is received within the current timeout window. This means that so long as a node is constantly receiving messages, the node will never timeout and enter the RoundChange state. This is a problem because, in some states, it allows a single malicious node to flood the message queue of another validator and trap that validator in its current state indefinitely. So far, we have found two ways this can occur in the `runCycle()` loop.

The first can be triggered in `runValidateState()`:

```go
func (i *Ibft) runValidateState() {
    ...
    for i.getState() == ValidateState {
        msg, ok := i.getNextMessage(timeout)

        ... // Move to RoundChangeState if msg == nil

      switch msg.Type {
    case proto.MessageReq_Prepare:
      i.state.addPrepared(msg)

    case proto.MessageReq_Commit:
      i.state.addCommitted(msg)

    default:
      panic(fmt.Sprintf("BUG: %s", reflect.TypeOf(msg.Type)))
    }

    if i.state.numPrepared() > i.state.NumValid() {
      // we have received enough pre-prepare messages
      sendCommit()
    }

    if i.state.numCommitted() > i.state.NumValid() {
      // we have received enough commit messages
      sendCommit()

      // try to commit the block (TODO: just to get out of the loop)
      i.setState(CommitState)
    }
  }
  ...
}
```

Here, the loop only exits when either

1. Enough commit messages are received, or
2. There are no new messages to process

If a single malicious validator repeatedly sends (for example) prepare messages, any non-malicious validator could indefinitely execute this loop without making any progress.

A similar potential vulnerability can be found in `runAcceptState()`:

```go
func (i *Ibft) runAcceptState() {
    ...
    for i.getState() == AcceptState {
    msg, ok := i.getNextMessage(timeout)

        ... // Move to RoundChangeState if msg == nil

    if msg.From != i.state.proposer.String() {
      i.logger.Error("msg received from wrong proposer")

      continue
    }
    ...
}
```

When a malicious validator that is *not* the proposer sends messages to a validator that is in the accept state, it will hit this `continue` statement and continue exiting the loop. When this validator only receives messages from the malicious node, then the non-malicious validator will become trapped in the AcceptState.

Additionally, in the case where the malicious node is the proposer, the loop can also be continuously executed when the following hook generates an error other than `errBlockVerificationFailed`:

```go
func (i *Ibft) runAcceptState() {
    ...
    for i.getState() == AcceptState {
        ...
        if hookErr := i.runHook(VerifyBlockHook, block.Number(), block); hookErr != nil {
          if errors.As(hookErr, &errBlockVerificationFailed) {
          i.logger.Error(...)
          i.handleStateErr(errBlockVerificationFailed)
        } else {
          i.logger.Error(fmt.Sprintf("Unable to run hook %s, %v", VerifyBlockHook, hookErr))
        }

        continue
      }
      ...
  }
}
```

Exploit Scenario To trigger the bug that fixes all non-malicious nodes in the ValidateState, consider the case where there is exactly one malicious node, and all nodes execute normally until reaching the ValidateState. At this point, assume all previous messages are dropped. Then, the malicious node should repeatedly send prepare messages. Each non-malicious node will execute `i.state.addPrepared(msg)` upon receiving the message, but since `i.state.numPrepared()` counts the number of prepare messages sent from *distinct* validators, `sendCommit()` will never execute. Similarly, since the malicious node is repeatedly sending new messages, all other validators will never timeout. Thus, a denial-of-service will be achieved.

Recommendation In the short term, implement a timeout for the functions `runSyncState()`, `runValidateState()`, and `runAcceptState()` such that the validator transitions to the RoundChange state whenever the timeout is reached.

In the long term, consider ignoring repeat messages to minimize the influence that malicious nodes can have on the system.

5.3.4 Failing Preparing-to-Ready transition

Severity	High	**Difficulty**	Medium
Type	Data Validation	**Finding ID**	V-DOGE-FV-006
Target	jury/consensus/ibft.go	**Status**	Open

Description Like subsection 5.3.2, malicious validators can take advantage of the fact that nodes process all valid messages. In `runValidateState()`:

```go
func (i *Ibft) runValidateState() {
    ...
    for i.getState() == ValidateState {
    msg, ok := i.getNextMessage(timeout)

        ... // Move to RoundChangeState if msg == nil

    switch msg.Type {
    case proto.MessageReq_Prepare:
      i.state.addPrepared(msg)

    case proto.MessageReq_Commit:
      i.state.addCommitted(msg)

    default:
      panic(fmt.Sprintf("BUG: %s", reflect.TypeOf(msg.Type)))
    }

    if i.state.numPrepared() > i.state.NumValid() {
      // we have received enough pre-prepare messages
      sendCommit()
    }
    ...
}
```

If a validator floods the message queue with Prepare messages, the transition to the Ready state will be delayed by processing all of these messages. An important note here is that `i.state.numPrepared()` counts the number of Prepare messages from *distinct* nodes, so the flood of messages from the malicious validator will not satisfy the `if` condition.

Exploit Scenario To trigger the bug, (1) ensure that some validator is in the ValidateState with no processed Prepare messages, and (2) have a malicious node repeatedly send Prepare messages of the correct sequence and round number.

Recommendation In the short term, avoid processing repeat messages.

In the long term, additional test cases should be written to show that whenever the threshold number of prepare messages is received in the ValidateState, the validator sends a commit message within a reasonable timeout.

6 Smart Contract Audit

We carried out an audit of the application contracts on commit `f8b31815e13b97d58da`. To evaluate the robustness of the Jury-contract, part of our audit focused on the detection of common security problems that appear in smart contracts. To perform this audit, we first identified a set of common bugs and vulnerabilities from existing databases and leveraged Vanguard, Veridise's static analyzer for blockchain applications; as well as open source static analysis tools such as Slither. In addition, we also performed manual code review. Table 6.1 summarizes a list of serious issues that are discovered by our tool and manual auditing.

Table 6.1: Summary of Smart Contract Audit.

ID	Description	Severity	Status
V-DOGE-CT-001	Governance double-spending	high	Fixed
V-DOGE-CT-002	ETH fee can't be recovered	high	Open
V-DOGE-CT-003	Lack of access control	Moderate	Open
V-DOGE-CT-004	Signers can be invalidated	Low	Open
V-DOGE-CT-005	Ghost validators	Low	Open
V-DOGE-CT-006	Potential reentrancy	Moderate	Fixed
V-DOGE-CT-007	Unchecked IGovernor cast	Low	Open
V-DOGE-CT-008	No cool down period on PoS	Low	Open

6.1 Detailed Description of Issues

In this section, we provide a detailed description of each issue.

6.1.1 Bug in Governance contract leads to double-spending attack

Severity	High	**Difficulty**	Easy
Type	Double-Spending	**Finding ID**	V-DOGE-CT-001
Target	contracts/Governance.sol	**Status**	Fixed

Description In the `claim()` method, the sender's total spend is set to their remaining amount of tokens to be earned. However, this behavior is inconsistent with the idea that the sender should have "spent" everything after claiming all tokens.

```
function claim() external {
  uint256 balance = getBalance(msg.sender);
  require(balance > 0, "Insufficient Balance");
  _whitelist[msg.sender].spend = balance;
  IGovernance(address(this)).safeTransfer(msg.sender, balance);
  emit claimed(msg.sender, balance);
}

```

```
function getBalance(address admin) view public returns (uint256) {
  return _whitelist[admin].balance.sub(_whitelist[admin].spend);
}
```

Exploit Scenario Suppose a malicious token has a balance of `b`. To trigger the bug, a malicious token can invoke `withdraw(b - 1)` to withdraw their current balance minus 1, so that their spend becomes `b - 1`. This results a transfer of `b - 1` tokens from the governance contract to the sender. Then, they can invoke `claim()` to reset their spend to `b - (b - 1) = 1`. Because their spend is now reset to 1, they are free to continue calling `withdraw` and `claim` until all tokens have been drained from the governance contract.

Recommendation The calculation of spend should be corrected:

```
  _whitelist[msg.sender].spend = _whitelist[msg.sender].balance;
```

This bug appears to be the result of flawed software engineering practices. To prevent similar bugs from occurring in the future, we recommend the developers adopt the following practices:

- Document each method with a clear description of what the method should do conceptually.
- To avoid confusion, use clear terminology when naming methods and variables. We believe that this bug likely resulted from a developer confusing the meaning of `getBalance()` with the `.balance` field of the `User` struct. Furthermore, it is also easy to confuse the ERC20 methods and terminology with the "balances" defined by `getBalance()` and the `User` struct.
  As an example, the following methods and fields can be renamed as follows to improve clarity:
  - `getBalance()`: rename to `getRemainingClaims()`.
  - `User.spend`: rename to `claimed`.
  - `User.balance`: rename to `maxClaims`.

6.1.2 ETH fee cannot be recovered from Bridge contract

Severity	High	**Difficulty**	Easy
Type	Locked Funds	**Finding ID**	V-DOGE-CT-002
Target	contracts/Bridge.sol	**Status**	Open

Description Although the bridge contract allows ETH payments to be made to the `withdraw()` method, there are no methods that allow ETH to be retrieved from the bridge contract. This does not seem to be the intention of the developers, as the `withdraw()` method takes a "fee" from the ETH payment that is not added to the `_totalSupply` variable.

Recommendation Unless the developers intend for the `withdraw()` method to act as a "black hole" for ETH, they should add a method that allows authorized accounts to transfer ETH away from the contract. This type of issue can also be detected by static analysis tools, which we recommend the developers incorporate into their workflow.

6.1.3 Bridge withdraw method has no access controls

Severity	Medium	**Difficulty**	Medium
Type	Access Control	**Finding ID**	V-DOGE-CT-003
Target	contracts/Bridge.sol	**Status**	Open

Description The `withdraw()` method has no access controls, despite the `deposit()` method having strong access controls. Specifically:

- The `deposit()` method requires a simple majority of the signers in order for an `Order` to be approved for a `receiver`, while `withdraw()` does not check any properties involving the `receiver`.
- In `deposit()`, the `_totalSupply` variable can only be increased if an order is approved in `deposit()`. However, anybody can decrease `_totalSupply` as long as they pay enough ETH to `withdraw()`.

Exploit Scenario A wealthy Ethereum account can call `withdraw()` with `_totalSupply * 1000 / (1000 - _rate)` wei to reduce `_totalSupply` to zero. Furthermore, they are able to choose any `receiver` to be used in the emitted `Withdrawn` event.

Recommendation The developers should enforce some sort of access controls in `withdraw()` by checking properties of the `receiver` parameter. For example, they may want to introduce a mapping that tracks the total deposits made by each receiver. The value of `receiver` in the mapping can be increased in `deposit()` and decreased in `withdraw()`. The developers could add a `require` statement to the beginning of `withdraw()` that checks that the receiver is able to withdraw that amount.

6.1.4 Bridge order signers can be invalidated

Severity	Low	**Difficulty**	Medium
Type	Data Validation	**Finding ID**	V-DOGE-CT-004
Target	contracts/Bridge.sol	**Status**	Open

Description The bridge contract allows the owner to remove signers with the `deleteSigner()` method. However, this does not update remove those signers from any existing unapproved orders. Because the `deposit()` method only checks that the size of an order's signers array exceeds half of the size of the current signers set, this means that signers that have been removed may still count as "votes" for the simple majority of current signers:

```
    if (order.signers.length > _signers.length/2 && !order.finished) {
      order.finished = true;
      _totalSupply = _totalSupply.add(amount);
      emit Deposited(order.receiver, order.amount, order.txid, order.sender);
    }
```

For example, if there are currently two signers C, D but an order has been approved by signers A, B that were previously removed, then signer C calling `deposit()` on the order would cause the order to be finished, even though only one of the two current signers have approved the order.

Exploit Scenario We believe that it would be difficult, but still possible, to exploit this bug. If a block includes transactions where 1) an owner removes a signer; 2) an owner adds a signer; and 3) the signer being removed approves an order so that it is 1 signer away from simple majority, then a malicious miner may reorder the 2nd and 3rd transactions to ensure that the order is approved before requiring the approval of the new signer. The consequences of such a reordering are highly situational, however.

Recommendation If this is not the behavior intended by the developers, or if it is and the developers nevertheless want to eliminate the possibility of such an exploit ever occurring, we recommend that the developers insert code in the `deposit()` method to count the number of valid signers. This count should be used for the majority check instead of `order.signers.length`.

6.1.5 Missing check allows creation of "ghost validators"

Severity	Low	**Difficulty**	Easy
Type	Data Validation	**Finding ID**	V-DOGE-CT-005
Target	contracts/ValidatorSet.sol	**Status**	Open

Description `addValidator()` will insert the provided address into the `_validators` array. However, if the provided address is already a validator, then the address will be inserted into `_validators` again even if it is already in the array.

```
function addValidator(address account) external onlyOwner {
  require(_addressToStakedAmount[account] >= _threshold, "Account must be staked enough");
  _addressToValidatorIndex[account] = _validators.length;
  _addressToIsValidator[account] = true;
  _validators.push(account);
  emit ValidatorAdded(msg.sender, account);
}
```

Exploit Scenario The `_owner` may invoke `addValidator()` by accident, which inserts a validator multiple times into the `_validators` array. Then, even if the `_owner` invokes `deleteValidator()` to remove that validator, the validator will remain in the `_validators` array. An account that then calls the `validators()` method will obtain an incorrect list of validators.

Recommendation To fix this bug, insert a require statement to check validator status:

```
function addValidator(address account) external onlyOwner {
  require(_addressToStakedAmount[account] >= _threshold, "Account must be staked enough");
  require(!_addressToIsValidator[account], "Account cannot already be a validator");
  _addressToValidatorIndex[account] = _validators.length;
```

```
    _addressToIsValidator[account] = true;
    _validators.push(account);
    emit ValidatorAdded(msg.sender, account);
  }
```

To avoid similar issues in future projects, the developers may want to consider using a well-tested and audited set data structure library, such as OpenZeppelin's EnumerableSet library.

6.1.6 Potential Reentrancy issues

Severity	Moderate	**Difficulty**	Medium
Type	Reentrancy	**Finding ID**	V-DOGE-CT-006
Target	See description	**Status**	Open

Description We ran the Slither static analyzer on the DogeChain application contracts to look for issues. The tool reported multiple reentrancy issues on almost all of the contracts. Most of the issues are due to events being emitted after external calls, allowing an attacker to reorder events by using reentrancy.

After reviewing the results, we identified the following reported issues to be of concern:

```
Reentrancy in ValidatorSet.stake(uint256) (contracts/ValidatorSet.sol#80-86)
Reentrancy in ValidatorSet.unstake() (contracts/ValidatorSet.sol#88-103)
```

We believe these have the potential to be problematic if a malicious owner sets a malicious governor contract:

- In `stake()`, the `Staked` event is emitted after the transfer, so a reentrancy attack can be used to reorder the `Staked` events.
- In `unstake()`, it may be possible to construct a reentrancy attack in which the governor token has withdrawn funds from the ValidatorSet contract, but the sender has not yet been removed as a validator.

Assuming the governance token can be trusted, we identified the following reported issues as false positives:

```
Reentrancy in Governance.claim() (contracts/Governance.sol#50-56)
Reentrancy in Governance.withdraw(uint256) (contracts/Governance.sol#58-63)
Reentrancy in MerkleDistributor.claim(uint256,address,uint256,bytes32[]) (contracts/
    MerkleDistributor.sol#36-50)
Reentrancy in StakedReward.unstake() (contracts/StakedReward.sol#107-124)
Reentrancy in StakedReward.changePerBlock(uint256) (contracts/StakedReward.sol
    #142-147)
Reentrancy in StakedReward.claim() (contracts/StakedReward.sol#126-140)
Reentrancy in StakedReward.stake() (contracts/StakedReward.sol#92-105)
Reentrancy in StakedReward.updateReward() (contracts/StakedReward.sol#37-48)
Reentrancy in TimeLock.create(address,uint256,uint256,uint256) (contracts/TimeLock.
    sol#30-38)
Reentrancy in TimeLock.unlock() (contracts/TimeLock.sol#51-67)
```

We believe the issues are false positives for the following reasons:

- In `Governance.sol`, the reentrant calls are to its own ERC20 methods, which do not make any further calls.
- In `StakedReward`, the detected calls are due to transfers to either externally owned accounts (which cannot make reentrant calls) or to what is assumed to be the governance token.
- In `MerkleDistributor.sol` and `TimeLock`, the detected calls are to what is assumed to be the governance token.

We again emphasize that our analysis relies on the assumption that the governance token can be trusted, and that it corresponds to the source code in `Governance.sol`. Consequently, these issues may no longer be false positives if `Governance.sol` is modified to make additional external calls.

Exploit Scenario Suppose the owner is malicious, so that they initialize the contracts with a malicious governance token or that they change the governance token to a malicious one.

A malicious governance token may use reentrant calls to manipulate the order of events, so that any off-chain applications that monitor the event log will be subject to this manipulated ordering.

It may also be possible to construct a reentrancy attack that abuses the behavior of `ValidatorSet .unstake()`, although we did not construct a concrete scenario in which such behavior can exploited.

Recommendation Although we identified most of the reentrancy issues reported by Slither as false positives, our reasoning relies on assumptions that the governance token can be trusted. Future changes to the code may invalidate such assumptions. Therefore, we suggest that the developers should address the reentrancy issues by ensuring that all events are emitted and all state variable updates are performed before making external method calls. The developers should then re-run Slither on their contracts to ensure that the issues have been fixed.

6.1.7 Unchecked casts to IGovernor interface

Severity	Low	**Difficulty**	Medium
Type	Data Validation	**Finding ID**	V-DOGE-CT-007
Target	Governance.sol	**Status**	Open

Description Multiple contracts take in an address that is later cast to IGovernor. However, none of these contracts check that the address satisfies the IGovernor interface. If these contracts are deployed with the address set to a non-IGovernor contract, perhaps by mistake, then these contracts will not function correctly.

```
function claim() external {
  uint256 balance = getBalance(msg.sender);
  require(balance > 0, "Insufficient Balance");
  _whitelist[msg.sender].spend = _whitelist[msg.sender].spend.add(balance);
  IGovernance(address(this)).safeTransfer(msg.sender, balance);
  emit claimed(msg.sender, balance);
}
```

```

function withdraw(uint256 amount) external {
  require(getBalance(msg.sender) >= amount, "Insufficient Balance");
  _whitelist[msg.sender].spend = _whitelist[msg.sender].spend.add(amount);
  IGovernance(address(this)).safeTransfer(msg.sender, amount);
  emit Withdrawn(msg.sender, amount);
}
```

Recommendation To improve compile time type safety, the casts should occur when the address is set, such as in the constructor or in any `setXY()` methods (note, however, that the casts are merely assumptions and do not actually check that the interface holds). It may also be possible to insert code to dynamically check that the provided address satisfies the IGovernor interface, although such code would be high effort to write.

6.1.8 Staking for PoS with no cool down period

Severity	Low	**Difficulty**	Easy
Type	Data Validation	**Finding ID**	V-DOGE-CT-007
Target	validatorset.sol	**Status**	Open

Description Proof of stake mechanism allows any nodes to join the validator set as long as they have staked an arbitrary amount of token and approved by the contract owner, of which the logic is implemented using a smart contract `validatorset.sol`. The consensus module sync the validator set by querying the smart contract after each epoch and recognize all nodes in validator set at that time as validators for next epoch. Given that the smart contract also allows for unstaking immediately after staking, one could stake before one epoch ends and unstake immediately after next epochs start. By doing so, the staking amount only requires to be in position by one for a few seconds while they can make their nodes as a validators for next epoch (by default 20,000 seconds).

```
function unstake() external onlyEOA {
  uint256 amount = _addressToStakedAmount[msg.sender];
  require(amount > 0, "Only staker can call function");

  _addressToStakedAmount[msg.sender] = 0;
  _stakedAmount = _stakedAmount.sub(amount);

  IERC20(_token).transfer(msg.sender, amount);

  if (_addressToIsValidator[msg.sender]) {
    require(_validators.length > _minimum, "Validators can't be less than minimum");
    _deleteFromValidators(msg.sender);
  }

  emit Unstaked(msg.sender, amount);
}

```

Exploit Scenario An attacker can easily make their nodes to significantly outnumber the legit nodes in the validator set with extremely low cost. With more faulty validators than legit validators in the network, no new block would be formed and no new transaction could be processed. An attacker could also conduct 51% attack, allowing them to manipulate transactions in the new blocks.

Recommendations Contract owner should evaluate the staker carefully before adding them to the validator set. In the meantime, consider adding a cool down period in the staking contract so that the users could not immediately unstake after staking.