

Assure DeFi[®]

THE VERIFICATION **GOLD STANDARD**



Security Assessment

PRIVIX

Date: 27/06/2025

Audit Status: PASS





Audit Edition: Code audit



ASSURE DEFI[®]
THE VERIFICATION **GOLD STANDARD**

Risk Analysis

Vulnerability summary

Classification	Description
 High	Vulnerabilities that lead to direct compromise of critical assets, large-scale data exposure, unauthorized fund transfers, or full system takeover.
 Medium	Flaws that weaken security posture or privacy but do not immediately enable catastrophic failures.
 Low	Issues that have minimal direct impact, often involving best-practice deviations or potential future risks.
 Informational	Observations, style concerns, or suggestions that do not constitute vulnerabilities but may improve security hygiene.

Scope

Target Code And Revision

Project	Assure
Language	GO
Codebase	https://github.com/PrivixAI-labs/Privix-node Commit: 6e1de94794419a080234136b512cdf1e682cb366 Fixed version: Commit: ea530da55d73586081edf3357b75406c292f06b2
Audit Methodology	Static, Manual

Detailed Technical Report



1. Consensus Safety Violation [FIXED ✓]

Location: consensus/pri-bft/validators.go

```
// CalcMaxFaultyNodes calculates the maximum number of faulty nodes
// ... N = 3F + 1 -> F = (N - 1) / 3
return (s.Len() - 1) / 2 // More fault tolerance
```

Issue: IBFT requires $F \leq (N-1)/3$ to guarantee safety; using $(N-1)/2$ allows up to 49% of nodes to be Byzantine, violating the $\frac{2}{3}$ majority assumption.

Exploit: An attacker controlling between $\frac{1}{3}$ and $\frac{1}{2}$ of validators can equivocate without being detected, split the network, or finalize conflicting blocks.

Remediation: Revert to the standard formula:

```
return (s.Len() - 1) / 3
```

Fix: Reverted to the standard formula.

2. Broken Proposer Selection [FIXED ✓]

Location: consensus/pri-bft/validators.go

```
func CalcProposer(...) {
    if lastProposer == types.ZeroAddress {
        seed = round
    } else {
        ... // compute offset
    }
    // ← Missing: `pick := seed % uint64(validators.Len()); return
    validators.At(pick)`
}
// ...
// pick := seed % uint64(validators.Len())
// return validators.At(pick)
```

Issue: The body computes a seed but never selects or returns a proposer. The correct logic is commented out, so CalcProposer returns the zero-value Validator, leading to deadlocks or unpredictable proposer behavior.

Exploit: Consensus stalls indefinitely or makes no progress, an attacker could trigger view-changes repeatedly.

Remediation: Restore the commented logic:

```
pick := seed % uint64(validators.Len())  
return validators.At(pick)
```

Fix: CalcProposer now correctly selects and returns a validator.

3. RPC Interface Exposed Without Authentication [FIXED ✓]

Location: server/server.go & server/builtin.go

```
// e.g. Listen on 0.0.0.0:8545  
http.ListenAndServe(cfg.JSONRPCAddr, mux)
```

Issue: JSON-RPC is bound to all interfaces with no authentication or IP whitelisting..

Exploit: Remote attackers can invoke administrative RPCs (eth_sendTransaction, admin_*), drain funds, or DOS the node.

Remediation: Bind RPC to localhost by default.

Implement token-based auth or TLS client certificates.

Sanitize and limit available methods via an allowlist.

Fix: Loopback binding: In command/helper/helper.go (lines 206–209), the default JSON-RPC listen address is now set to 127.0.0.1:8545 instead of 0.0.0.0:8545, Token-based middleware:

In jsonrpc/jsonrpc.go (lines 127–135), every incoming RPC request is passed through an AuthToken checker and an explicit method allowlist and lastly TLS Client-Cert Support:

The server initialization in server/server.go now conditionally enables TLS based on config flags, requiring client certificates if rpc.tls.clientAuth=true

4. Libp2p Flood & Message-Handling Risks [FIXED ✓]

Location: network/ package

Issue: No limits on inbound message size, no per-peer rate limiting or backoff. Malformed or oversized messages may exhaust memory or CPU.

Exploit: A malicious peer floods the gossip protocol, triggering OOM or high CPU, splitting the network.

Remediation: Enable libp2p built-in flood protection (fsm.AddrsFactory, BasicConnectionLimiter).

Validate and bound message sizes before unmarshalling.

Implement per-peer rate limits and blacklisting on misbehavior.

Fix: The next measures establish fixed bounds on message sizes, queues, and total connections:

gossipsub hardening;

pubsub.WithMaxMessageSize(maxGossipMessageSize) to reject oversized messages

pubsub.WithStrictSignatureVerification(true) to drop malformed or spoofed messages

pubsub.WithValidateQueueSize(validateBufferSize) &

pubsub.WithPeerOutboundQueueSize(peerOutboundBufferSize) to cap in-memory queues
connection management;
libp2p.AddrsFactory(addrsFactory) to control listen/dial addresses
libp2p.ConnectionManager(connManager) (via connmgr.NewConnManager) to limit total and idle peer connections



1. Predictable Random IDs in CLI [FIXED ✓]

Location: command/default.go

```
rand.Seed(time.Now().UnixNano())
// ...
id := 1000 + rand.Uint64() % (1<<32 - 1000)
```

Issue: Using math/rand seeded with time makes generated IDs predictable, enabling attackers to precompute and target upcoming IDs.

Exploit: An attacker can guess future IDs for temporary resources or session tokens and hijack them.

Remediation: Switch to crypto/rand for all security-sensitive randomness:

```
var idBuf [8]byte
if _, err := crand.Read(idBuf[:]); err != nil { ... }
id := binary.BigEndian.Uint64(idBuf[:])
```

Fix: No more math/rand in command/default.go (or anywhere under command/); the rand.Seed(...) call and rand.Uint64() usages are gone.

Instead, ID generation now works like this:

```
var buf [8]byte
if _, err := crand.Read(buf[:]); err != nil {
    return fmt.Errorf("failed to generate token: %w", err)
}
id := binary.BigEndian.Uint64(buf[:])
```

All other CLI tools that need randomness have been added to use crypto/rand as well.

2. Insecure Local Secrets on Filesystem [FIXED ✓]

Location: secrets/secrets

```
// LocalFSManager stores secrets as plaintext in ~/.privix/keystore/
ioutil.WriteFile(path, data, 0644)
```

Issue: Secrets (private keys, tokens) are written unencrypted with world-readable permissions by default.

Exploit: Any local user or malware can read private keys and compromise node identity.

Remediation:

Encrypt on-disk secrets with a user-provided passphrase (for example using `scrypt + AES-GCM`).

Enforce 0600 file permissions.

Document proper OS-level hardening.

Fix: In `secrets/local/local.go` (around line 171), the `LocalFSManager` now wraps all secret writes in AES-GCM encryption using a user-supplied passphrase additionally, after encryption, files are written with 0600 mode via `and` the CLI's `init` keystore help text now explicitly calls out the passphrase prompt and OS-level hardening recommendations

3. LevelDB: Lack of Atomicity & Permission Hardening [FIXED ✓]

Location: `blockchain/storage/leveldb/backend.go`

Issue: No explicit use of `leveldb.OpenFile` options for write-ahead logging or snapshotting, no `fsync` nor file-mode checks. Race conditions during crash recovery may cause data corruption.

Exploit: Node crash at a critical point can lead to chain fork, state corruption, or replay of partially-written blocks.

Remediation:

Open LevelDB with `Options{Sync: true}`.

Use `FileMode: 0600`.

Add background compaction and snapshot tests to simulate crash recovery.

Fix: In `blockchain/storage/leveldb/leveldb.go` (around line 63) the network now open the DB with:

```
db, err := leveldb.OpenFile(path, &opt.Options{
    WriteOptions: opt.WriteOptions{Sync: true},
    // ...
})
```

so writes are fsynced to disk as recommended. As an additional recommendation, there is no `FileMode: 0o600` (or equivalent) in your `opt.Options`, so LevelDB data files are still created with default perms (typically world-readable), we recommend to add file-mode locking



No low issues were found.



1. No Vendor/Dependency Vulnerability Scan [Acknowledge]

Location: go.mod

Risk: Transitive dependency with known CVE can introduce remote code execution or supply-chain attacks.

Remediation:

Integrate govulncheck into CI (make govulncheck).

Pin dependencies explicitly in go.mod

Regularly run go mod tidy and review updated CVEs.

Testing [GO]

Blockchain:

```
package blockchain

import (
    "errors"
    "fmt"
    "math/big"
    "reflect"
    "testing"

    "github.com/hashicorp/go-hclog"
    lru "github.com/hashicorp/golang-lru"
    "github.com/PrivixAI-labs/Privix-node/helper/common"
    "github.com/PrivixAI-labs/Privix-node/helper/hex"
    "github.com/PrivixAI-labs/Privix-node/state"

    "github.com/PrivixAI-labs/Privix-node/chain"
    "github.com/stretchr/testify/assert"
    "github.com/stretchr/testify/require"

    "github.com/PrivixAI-labs/Privix-node/blockchain/storage"
    "github.com/PrivixAI-labs/Privix-node/blockchain/storage/memory"
    "github.com/PrivixAI-labs/Privix-node/types"
)

func TestGenesis(t *testing.T) {
    b := NewTestBlockchain(t, nil)

    // add genesis block
    genesis := &types.Header{Difficulty: 1, Number: 0}
    genesis.ComputeHash()

    assert.NoError(t, b.writeGenesisImpl(genesis))

    header := b.Header()
    assert.Equal(t, header.Hash, genesis.Hash)
}
```

```

type dummyChain struct {
    headers map[byte]*types.Header
}

func (c *dummyChain) add(h *header) error {
    if _, ok := c.headers[h.hash]; ok {
        return fmt.Errorf("hash already imported")
    }

    var parent types.Hash
    if h.number != 0 {
        p, ok := c.headers[h.parent]
        if !ok {
            return fmt.Errorf("parent not found %v", h.parent)
        }

        parent = p.Hash
    }

    hh := &types.Header{
        ParentHash: parent,
        Number:     h.number,
        Difficulty: h.diff,
        ExtraData:  []byte{h.hash},
    }

    hh.ComputeHash()
    c.headers[h.hash] = hh

    return nil
}

type header struct {
    hash    byte
    parent  byte
    number  uint64
    diff    uint64
}

func (h *header) Parent(parent byte) *header {
    h.parent = parent
    h.number = uint64(parent) + 1

    return h
}

```

```

func (h *header) Diff(d uint64) *header {
    h.diff = d

    return h
}

func (h *header) Number(d uint64) *header {
    h.number = d

    return h
}

func mock(number byte) *header {
    return &header{
        hash:    number,
        parent:  number - 1,
        number:  uint64(number),
        diff:    uint64(number),
    }
}

func TestInsertHeaders(t *testing.T) {
    type evnt struct {
        NewChain []*header
        OldChain []*header
        Diff     *big.Int
    }

    type headerEvt struct {
        header *header
        event  *evnt
    }

    var cases = []struct {
        Name      string
        History   []*headerEvt
        Head      *header
        Forks     []*header
        Chain     []*header
        TD        uint64
    }{
        {
            Name: "Genesis",
            History: []*headerEvt{

```

```

        {
            header: mock(0x0),
        },
    },
    Head: mock(0x0),
    Chain: []*header{
        mock(0x0),
    },
    TD: 0,
},
{
    Name: "Linear",
    History: []*headerEvt{
        {
            header: mock(0x0),
        },
        {
            header: mock(0x1),
            event: &evnt{
                NewChain: []*header{
                    mock(0x1),
                },
                Diff: big.NewInt(1),
            },
        },
        {
            header: mock(0x2),
            event: &evnt{
                NewChain: []*header{
                    mock(0x2),
                },
                Diff: big.NewInt(3),
            },
        },
    },
    Head: mock(0x2),
    Chain: []*header{
        mock(0x0),
        mock(0x1),
        mock(0x2),
    },
    TD: 0 + 1 + 2,
},
{
    Name: "Keep block with higher difficulty",

```

```

History: []*headerEvt{
    {
        header: mock(0x0),
    },
    {
        header: mock(0x1),
        event: &evnt{
            NewChain: []*header{
                mock(0x1),
            },
            Diff: big.NewInt(1),
        },
    },
    {
        header: mock(0x3).Parent(0x1).Diff(5),
        event: &evnt{
            NewChain: []*header{
                mock(0x3).Parent(0x1).Diff(5),
            },
            Diff: big.NewInt(6),
        },
    },
    {
        // This block has lower difficulty than the current chain (fork)
        header: mock(0x2).Parent(0x1).Diff(3),
        event: &evnt{
            OldChain: []*header{
                mock(0x2).Parent(0x1).Diff(3),
            },
        },
    },
},
Head: mock(0x3),
Forks: []*header{mock(0x2)},
Chain: []*header{
    mock(0x0),
    mock(0x1),
    mock(0x3).Parent(0x1).Diff(5),
},
TD: 0 + 1 + 5,
},
{
    Name: "Reorg",
    History: []*headerEvt{
        {

```

```

        header: mock(0x0),
    },
    {
        header: mock(0x1),
        event: &evnt{
            NewChain: []*header{
                mock(0x1),
            },
            Diff: big.NewInt(1),
        },
    },
    {
        header: mock(0x2),
        event: &evnt{
            NewChain: []*header{
                mock(0x2),
            },
            Diff: big.NewInt(1 + 2),
        },
    },
    {
        header: mock(0x3),
        event: &evnt{
            NewChain: []*header{
                mock(0x3),
            },
            Diff: big.NewInt(1 + 2 + 3),
        },
    },
    {
        // First reorg
        header: mock(0x4).Parent(0x1).Diff(10).Number(2),
        event: &evnt{
            // add block 4
            NewChain: []*header{
                mock(0x4).Parent(0x1).Diff(10).Number(2),
            },
            // remove block 2 and 3
            OldChain: []*header{
                mock(0x2),
                mock(0x3),
            },
            Diff: big.NewInt(1 + 10),
        },
    },

```



```

    {
        header: mock(0x5).Parent(0x4).Diff(11).Number(3),
        event: &evnt{
            NewChain: []*header{
                mock(0x5).Parent(0x4).Diff(11).Number(3),
            },
            Diff: big.NewInt(1 + 10 + 11),
        },
    },
    {
        header: mock(0x6).Parent(0x3).Number(4),
        event: &evnt{
            // Lower difficulty, its a fork
            OldChain: []*header{
                mock(0x6).Parent(0x3).Number(4),
            },
        },
    },
},
Head: mock(0x5),
Forks: []*header{mock(0x6)},
Chain: []*header{
    mock(0x0),
    mock(0x1),
    mock(0x4).Parent(0x1).Diff(10).Number(2),
    mock(0x5).Parent(0x4).Diff(11).Number(3),
},
TD: 0 + 1 + 10 + 11,
},
{
    Name: "Forks in reorgs",
    History: []*headerEvt{
        {
            header: mock(0x0),
        },
        {
            header: mock(0x1),
            event: &evnt{
                NewChain: []*header{
                    mock(0x1),
                },
                Diff: big.NewInt(1),
            },
        },
    },
    {

```

```

        header: mock(0x2),
        event: &evnt{
            NewChain: []*header{
                mock(0x2),
            },
            Diff: big.NewInt(1 + 2),
        },
    },
    {
        header: mock(0x3),
        event: &evnt{
            NewChain: []*header{
                mock(0x3),
            },
            Diff: big.NewInt(1 + 2 + 3),
        },
    },
    {
        // fork 1. 0x1 -> 0x2 -> 0x4
        header: mock(0x4).Parent(0x2).Diff(11),
        event: &evnt{
            NewChain: []*header{
                mock(0x4).Parent(0x2).Diff(11),
            },
            OldChain: []*header{
                mock(0x3),
            },
            Diff: big.NewInt(1 + 2 + 11),
        },
    },
    {
        // fork 2. 0x1 -> 0x2 -> 0x3 -> 0x5
        header: mock(0x5).Parent(0x3),
        event: &evnt{
            OldChain: []*header{
                mock(0x5).Parent(0x3),
            },
        },
    },
    {
        // fork 3. 0x1 -> 0x2 -> 0x6
        header: mock(0x6).Parent(0x2).Diff(5),
        event: &evnt{
            OldChain: []*header{
                mock(0x6).Parent(0x2).Diff(5),
            },
        },
    },

```

```

        },
    },
},
Head: mock(0x4),
Forks: []*header{mock(0x5), mock(0x6)},
Chain: []*header{
    mock(0x0),
    mock(0x1),
    mock(0x2),
    mock(0x4).Parent(0x2).Diff(11),
},
TD: 0 + 1 + 2 + 11,
},
{
    Name: "Head from old long fork",
    History: []*headerEvt{
        {
            header: mock(0x0),
        },
        {
            header: mock(0x1),
            event: &evnt{
                NewChain: []*header{
                    mock(0x1),
                },
                Diff: big.NewInt(1),
            },
        },
        {
            header: mock(0x2),
            event: &evnt{
                NewChain: []*header{
                    mock(0x2),
                },
                Diff: big.NewInt(1 + 2),
            },
        },
        {
            // fork 1.
            header: mock(0x3).Parent(0x0).Diff(5),
            event: &evnt{
                NewChain: []*header{
                    mock(0x3).Parent(0x0).Diff(5),
                },
            },
        },
    },
}

```

```

        OldChain: []*header{
            mock(0x1),
            mock(0x2),
        },
        Diff: big.NewInt(0 + 5),
    },
},
{
    // Add back the 0x2 fork
    header: mock(0x4).Parent(0x2).Diff(10),
    event: &evnt{
        NewChain: []*header{
            mock(0x4).Parent(0x2).Diff(10),
            mock(0x2),
            mock(0x1),
        },
        OldChain: []*header{
            mock(0x3).Parent(0x0).Diff(5),
        },
        Diff: big.NewInt(1 + 2 + 10),
    },
},
},
Head: mock(0x4).Parent(0x2).Diff(10),
Forks: []*header{
    mock(0x2),
    mock(0x3).Parent(0x0).Diff(5),
},
Chain: []*header{
    mock(0x0),
    mock(0x1),
    mock(0x2),
    mock(0x4).Parent(0x2).Diff(10),
},
TD: 0 + 1 + 2 + 10,
},
}

for _, cc := range cases {
    t.Run(cc.Name, func(t *testing.T) {
        b := NewTestBlockchain(t, nil)

        chain := dummyChain{
            headers: map[byte]*types.Header{},
        }
    })
}

```

```

for _, i := range cc.History {
    if err := chain.add(i.header); err != nil {
        t.Fatal(err)
    }
}

checkEvents := func(a []*header, b []*types.Header) {
    if len(a) != len(b) {
        t.Fatal("bad size")
    }
    for indx := range a {
        if chain.headers[a[indx].hash].Hash != b[indx].Hash {
            t.Fatal("bad")
        }
    }
}

// genesis is 0x0
if err := b.writeGenesisImpl(chain.headers[0x0]); err != nil {
    t.Fatal(err)
}

// we need to subscribe just after the genesis and history
sub := b.SubscribeEvents()

// run the history
for i := 1; i < len(cc.History); i++ {
    headers := []*types.Header{chain.headers[cc.History[i].header.hash]}
    if err := b.WriteHeadersWithBodies(headers); err != nil {
        t.Fatal(err)
    }

    // get the event
    evnt := sub.GetEvent()
    checkEvents(cc.History[i].event.NewChain, evnt.NewChain)
    checkEvents(cc.History[i].event.OldChain, evnt.OldChain)

    if evnt.Difficulty != nil {
        if evnt.Difficulty.Cmp(cc.History[i].event.Diff) != 0 {
            t.Fatal("bad diff in event")
        }
    }
}

head := b.Header()

```

```

expected, ok := chain.headers[cc.Head.hash]
assert.True(t, ok)

// check that we got the right hash
assert.Equal(t, head.Hash, expected.Hash)

forks, err := b.GetForks()
if err != nil && !errors.Is(err, storage.ErrNotFound) {
    t.Fatal(err)
}

expectedForks := []types.Hash{}

for _, i := range cc.Forks {
    expectedForks = append(expectedForks, chain.headers[i.hash].Hash)
}

if len(forks) != 0 {
    if len(forks) != len(expectedForks) {
        t.Fatalf("forks length dont match, expected %d but found %d",
len(expectedForks), len(forks))
    } else {
        if !reflect.DeepEqual(forks, expectedForks) {
            t.Fatal("forks dont match")
        }
    }
}

// Check chain of forks
if cc.Chain != nil {
    for indx, i := range cc.Chain {
        block, _ := b.GetBlockByNumber(uint64(indx), true)
        if block.Hash().String() != chain.headers[i.hash].Hash.String() {
            t.Fatal("bad")
        }
    }
}

if td, _ := b.GetChainTD(); cc.TD != td.Uint64() {
    t.Fatal("bad")
}
}))
}
}

```



```

func TestForkUnknownParents(t *testing.T) {
    b := NewTestBlockchain(t, nil)

    h0 := NewTestHeaders(10)
    h1 := AppendNewTestHeaders(h0[:5], 10)

    // Write genesis
    batchWriter := storage.NewBatchWriter(b.db)
    td := new(big.Int).SetUint64(h0[0].Difficulty)

    batchWriter.PutCanonicalHeader(h0[0], td)

    assert.NoError(t, b.writeBatchAndUpdate(batchWriter, h0[0], td, true))

    // Write 10 headers
    assert.NoError(t, b.WriteHeadersWithBodies(h0[1:]))

    // Cannot write this header because the father h1[11] is not known
    assert.Error(t, b.WriteHeadersWithBodies([]*types.Header{h1[12]}))
}

func TestBlockchainWriteBody(t *testing.T) {
    t.Parallel()

    var (
        addr = types.StringToAddress("1")
    )

    newChain := func(
        t *testing.T,
        txFromByTxHash map[types.Hash]types.Address,
        path string,
    ) *Blockchain {
        t.Helper()

        dbStorage, err := memory.NewMemoryStorage(nil)
        assert.NoError(t, err)

        chain := &Blockchain{
            db: dbStorage,
            txSigner: &mockSigner{
                txFromByTxHash: txFromByTxHash,
            },
        }
    }

```

```

        return chain
    }

    t.Run("should succeed if tx has from field", func(t *testing.T) {
        t.Parallel()

        tx := &types.Transaction{
            Value: big.NewInt(10),
            V:     big.NewInt(1),
            From:  addr,
        }

        block := &types.Block{
            Header: &types.Header{},
            Transactions: []*types.Transaction{
                tx,
            },
        }

        tx.ComputeHash(1)
        block.Header.ComputeHash()

        txFromByTxHash := map[types.Hash]types.Address{}

        chain := newChain(t, txFromByTxHash, "t1")
        defer chain.db.Close()
        batchWriter := storage.NewBatchWriter(chain.db)

        assert.NoError(
            t,
            chain.writeBody(batchWriter, block),
        )
        assert.NoError(t, batchWriter.WriteBatch())
    })

    t.Run("should return error if tx doesn't have from and recovering address fails",
func(t *testing.T) {
        t.Parallel()

        tx := &types.Transaction{
            Value: big.NewInt(10),
            V:     big.NewInt(1),
        }

```

```

    block := &types.Block{
        Header: &types.Header{},
        Transactions: []*types.Transaction{
            tx,
        },
    }

    tx.ComputeHash(1)
    block.Header.ComputeHash()

    txFromByTxHash := map[types.Hash]types.Address{}

    chain := newChain(t, txFromByTxHash, "t2")
    defer chain.db.Close()
    batchWriter := storage.NewBatchWriter(chain.db)

    assert.ErrorIs(
        t,
        errRecoveryAddressFailed,
        chain.writeBody(batchWriter, block),
    )
    assert.NoError(t, batchWriter.WriteBatch())
})

t.Run("should recover from address and store to storage", func(t *testing.T) {
    t.Parallel()

    tx := &types.Transaction{
        Value: big.NewInt(10),
        V:     big.NewInt(1),
    }

    block := &types.Block{
        Header: &types.Header{},
        Transactions: []*types.Transaction{
            tx,
        },
    }

    tx.ComputeHash(1)
    block.Header.ComputeHash()

    txFromByTxHash := map[types.Hash]types.Address{
        tx.Hash: addr,
    }

```

```

    chain := newChain(t, txFromByTxHash, "t3")
    defer chain.db.Close()
    batchWriter := storage.NewBatchWriter(chain.db)

    batchWriter.PutHeader(block.Header)

    assert.NoError(t, chain.writeBody(batchWriter, block))

    assert.NoError(t, batchWriter.WriteBatch())

    readBody, ok := chain.readBody(block.Hash())
    assert.True(t, ok)

    assert.Equal(t, addr, readBody.Transactions[0].From)
})
}

func Test_recoverFromFieldsInBlock(t *testing.T) {
    t.Parallel()

    var (
        addr1 = types.StringToAddress("1")
        addr2 = types.StringToAddress("1")
        addr3 = types.StringToAddress("1")
    )

    computeTxHashes := func(txs ...*types.Transaction) {
        for _, tx := range txs {
            tx.ComputeHash(1)
        }
    }

    t.Run("should succeed", func(t *testing.T) {
        t.Parallel()

        txFromByTxHash := map[types.Hash]types.Address{}
        chain := &Blockchain{
            txSigner: &mockSigner{
                txFromByTxHash: txFromByTxHash,
            },
        }

        tx1 := &types.Transaction{Nonce: 0, From: addr1}
        tx2 := &types.Transaction{Nonce: 1, From: types.ZeroAddress}
    })
}

```

```

computeTxHashes(tx1, tx2)

txFromByTxHash[tx2.Hash] = addr2

block := &types.Block{
    Transactions: []*types.Transaction{
        tx1,
        tx2,
    },
}

assert.NoError(
    t,
    chain.recoverFromFieldsInBlock(block),
)
})

t.Run("should stop and return error if recovery fails", func(t *testing.T) {
    t.Parallel()

    txFromByTxHash := map[types.Hash]types.Address{}
    chain := &Blockchain{
        txSigner: &mockSigner{
            txFromByTxHash: txFromByTxHash,
        },
    }

    tx1 := &types.Transaction{Nonce: 0, From: types.ZeroAddress}
    tx2 := &types.Transaction{Nonce: 1, From: types.ZeroAddress}
    tx3 := &types.Transaction{Nonce: 2, From: types.ZeroAddress}

    computeTxHashes(tx1, tx2, tx3)

    // returns only addresses for tx1 and tx3
    txFromByTxHash[tx1.Hash] = addr1
    txFromByTxHash[tx3.Hash] = addr3

    block := &types.Block{
        Transactions: []*types.Transaction{
            tx1,
            tx2,
            tx3,
        },
    }

```

```

    assert.ErrorIs(
        t,
        chain.recoverFromFieldsInBlock(block),
        errRecoveryAddressFailed,
    )

    assert.Equal(t, addr1, tx1.From)
    assert.Equal(t, types.ZeroAddress, tx2.From)
    assert.Equal(t, types.ZeroAddress, tx3.From)
})
}

func Test_recoverFromFieldsInTransactions(t *testing.T) {
    t.Parallel()

    var (
        addr1 = types.StringToAddress("1")
        addr2 = types.StringToAddress("1")
        addr3 = types.StringToAddress("1")
    )

    computeTxHashes := func(txs ...*types.Transaction) {
        for _, tx := range txs {
            tx.ComputeHash(1)
        }
    }

    t.Run("should succeed", func(t *testing.T) {
        t.Parallel()

        txFromByTxHash := map[types.Hash]types.Address{}
        chain := &Blockchain{
            logger: hclog.NewNullLogger(),
            txSigner: &mockSigner{
                txFromByTxHash: txFromByTxHash,
            },
        }

        tx1 := &types.Transaction{Nonce: 0, From: addr1}
        tx2 := &types.Transaction{Nonce: 1, From: types.ZeroAddress}

        computeTxHashes(tx1, tx2)

        txFromByTxHash[tx2.Hash] = addr2
    })
}

```



```

    transactions := []*types.Transaction{
        tx1,
        tx2,
    }

    assert.True(
        t,
        chain.recoverFromFieldsInTransactions(transactions),
    )
})

t.Run("should succeed even though recovery fails for some transactions", func(t
*testing.T) {
    t.Parallel()

    txFromByTxHash := map[types.Hash]types.Address{}
    chain := &Blockchain{
        logger: hclog.NewNullLogger(),
        txSigner: &mockSigner{
            txFromByTxHash: txFromByTxHash,
        },
    }

    tx1 := &types.Transaction{Nonce: 0, From: types.ZeroAddress}
    tx2 := &types.Transaction{Nonce: 1, From: types.ZeroAddress}
    tx3 := &types.Transaction{Nonce: 2, From: types.ZeroAddress}

    computeTxHashes(tx1, tx2, tx3)

    // returns only addresses for tx1 and tx3
    txFromByTxHash[tx1.Hash] = addr1
    txFromByTxHash[tx3.Hash] = addr3

    transactions := []*types.Transaction{
        tx1,
        tx2,
        tx3,
    }

    assert.True(t, chain.recoverFromFieldsInTransactions(transactions))

    assert.Equal(t, addr1, tx1.From)
    assert.Equal(t, types.ZeroAddress, tx2.From)
    assert.Equal(t, addr3, tx3.From)

```

```

}))

t.Run("should return false if all transactions has from field", func(t *testing.T) {
    t.Parallel()

    txFromByTxHash := map[types.Hash]types.Address{}
    chain := &Blockchain{
        logger: hclog.NewNullLogger(),
        txSigner: &mockSigner{
            txFromByTxHash: txFromByTxHash,
        },
    }

    tx1 := &types.Transaction{Nonce: 0, From: addr1}
    tx2 := &types.Transaction{Nonce: 1, From: addr2}

    computeTxHashes(tx1, tx2)

    txFromByTxHash[tx2.Hash] = addr2

    transactions := []*types.Transaction{
        tx1,
        tx2,
    }

    assert.False(
        t,
        chain.recoverFromFieldsInTransactions(transactions),
    )
})
}

func TestBlockchainReadBody(t *testing.T) {
    dbStorage, err := memory.NewMemoryStorage(nil)
    assert.NoError(t, err)

    txFromByTxHash := make(map[types.Hash]types.Address)
    addr := types.StringToAddress("1")

    b := &Blockchain{
        logger: hclog.NewNullLogger(),
        db:      dbStorage,
        txSigner: &mockSigner{
            txFromByTxHash: txFromByTxHash,
        },
    },

```

```

    }

    batchWriter := storage.NewBatchWriter(b.db)

    tx := &types.Transaction{
        Value: big.NewInt(10),
        V:     big.NewInt(1),
    }

    tx.ComputeHash(1)

    block := &types.Block{
        Header: &types.Header{},
        Transactions: []*types.Transaction{
            tx,
        },
    }

    block.Header.ComputeHash()

    txFromByTxHash[tx.Hash] = types.ZeroAddress

    batchWriter.PutCanonicalHeader(block.Header, big.NewInt(0))

    require.NoError(t, b.writeBody(batchWriter, block))

    assert.NoError(t, batchWriter.WriteBatch())

    txFromByTxHash[tx.Hash] = addr

    readBody, found := b.readBody(block.Hash())

    assert.True(t, found)
    assert.Equal(t, addr, readBody.Transactions[0].From)
}

func TestCalculateGasLimit(t *testing.T) {
    tests := []struct {
        name           string
        blockGasTarget uint64
        parentGasLimit uint64
        expectedGasLimit uint64
    }{
        {
            name:           "should increase next gas limit towards target",

```

```

        blockGasTarget: 25000000,
        parentGasLimit: 20000000,
        expectedGasLimit: 20000000/1024 + 20000000,
    },
    {
        name: "should decrease next gas limit towards target",
        blockGasTarget: 25000000,
        parentGasLimit: 26000000,
        expectedGasLimit: 26000000 - 26000000/1024,
    },
    {
        name: "should not alter gas limit when exactly the same",
        blockGasTarget: 25000000,
        parentGasLimit: 25000000,
        expectedGasLimit: 25000000,
    },
    {
        name: "should increase to the exact gas target if adding the delta
surpasses it",
        blockGasTarget: 25000000 + 25000000/1024 - 100, // - 100 so that it takes
less than the delta to reach it
        parentGasLimit: 25000000,
        expectedGasLimit: 25000000 + 25000000/1024 - 100,
    },
    {
        name: "should decrease to the exact gas target if subtracting the
delta surpasses it",
        blockGasTarget: 25000000 - 25000000/1024 + 100, // + 100 so that it takes
less than the delta to reach it
        parentGasLimit: 25000000,
        expectedGasLimit: 25000000 - 25000000/1024 + 100,
    },
}

for _, tt := range tests {
    t.Run(tt.name, func(t *testing.T) {
        storageCallback := func(storage *storage.MockStorage) {
            storage.HookReadHeader(func(hash types.Hash) (*types.Header, error) {
                return &types.Header{
                    // This is going to be the parent block header
                    GasLimit: tt.parentGasLimit,
                }, nil
            })
        }
    })
}

```

```

        b, blockchainErr := NewMockBlockchain(map[TestCallbackType]interface{}{
            StorageCallback: storageCallback,
        })
        if blockchainErr != nil {
            t.Fatalf("unable to construct the blockchain, %v", blockchainErr)
        }

        b.config.Params = &chain.Params{
            BlockGasTarget: tt.blockGasTarget,
        }

        nextGas, err := b.CalculateGasLimit(1)
        assert.NoError(t, err)
        assert.Equal(t, tt.expectedGasLimit, nextGas)
    })
}

// TestGasPriceAverage tests the average gas price of the
// blockchain
func TestGasPriceAverage(t *testing.T) {
    testTable := []struct {
        name           string
        previousAverage *big.Int
        previousCount   *big.Int
        newValues       []*big.Int
        expectedNewAverage *big.Int
    }{
        {
            "no previous average data",
            big.NewInt(0),
            big.NewInt(0),
            []*big.Int{
                big.NewInt(1),
                big.NewInt(2),
                big.NewInt(3),
                big.NewInt(4),
                big.NewInt(5),
            },
            big.NewInt(3),
        },
        {
            "previous average data",
            // For example (5 + 5 + 5 + 5 + 5) / 5
            big.NewInt(5),
        },
    }
}

```

```

        big.NewInt(5),
        []*big.Int{
            big.NewInt(1),
            big.NewInt(2),
            big.NewInt(3),
        },
        // (5 * 5 + 1 + 2 + 3) / 8
        big.NewInt(3),
    },
}

for _, testCase := range testTable {
    t.Run(testCase.name, func(t *testing.T) {
        // Setup the mock data
        blockchain := NewTestBlockchain(t, nil)
        blockchain.gpAverage.price = testCase.previousAverage
        blockchain.gpAverage.count = testCase.previousCount

        // Update the average gas price
        blockchain.updateGasPriceAvg(testCase.newValues)

        // Make sure the average gas price count is correct
        assert.Equal(
            t,
            int64(len(testCase.newValues))+testCase.previousCount.Int64(),
            blockchain.gpAverage.count.Int64(),
        )

        // Make sure the average gas price is correct
        assert.Equal(t, testCase.expectedNewAverage.String(),
            blockchain.gpAverage.price.String())
    })
}

// TestBlockchain_VerifyBlockParent verifies that parent block verification
// errors are handled correctly
func TestBlockchain_VerifyBlockParent(t *testing.T) {
    t.Parallel()

    emptyHeader := &types.Header{
        Hash:      types.ZeroHash,
        ParentHash: types.ZeroHash,
    }
    emptyHeader.ComputeHash()

```



```

t.Run("Missing parent block", func(t *testing.T) {
    t.Parallel()

    // Set up the storage callback
    storageCallback := func(storage *storage.MockStorage) {
        storage.HookReadHeader(func(hash types.Hash) (*types.Header, error) {
            return nil, errors.New("not found")
        })
    }

    blockchain, err := NewMockBlockchain(map[TestCallbackType]interface{}{
        StorageCallback: storageCallback,
    })
    if err != nil {
        t.Fatalf("unable to instantiate new blockchain, %v", err)
    }

    // Create a dummy block
    block := &types.Block{
        Header: &types.Header{
            ParentHash: types.ZeroHash,
        },
    }

    assert.ErrorIs(t, blockchain.verifyBlockParent(block), ErrParentNotFound)
})

t.Run("Parent hash mismatch", func(t *testing.T) {
    t.Parallel()

    // Set up the storage callback
    storageCallback := func(storage *storage.MockStorage) {
        storage.HookReadHeader(func(hash types.Hash) (*types.Header, error) {
            return emptyHeader, nil
        })
    }

    blockchain, err := NewMockBlockchain(map[TestCallbackType]interface{}{
        StorageCallback: storageCallback,
    })
    if err != nil {
        t.Fatalf("unable to instantiate new blockchain, %v", err)
    }

```

```

// Create a dummy block whose parent hash will
// not match the computed parent hash
block := &types.Block{
    Header: emptyHeader,
}

assert.ErrorIs(t, blockchain.verifyBlockParent(block), ErrParentHashMismatch)
})

t.Run("Invalid block sequence", func(t *testing.T) {
    t.Parallel()

    // Set up the storage callback
    storageCallback := func(storage *storage.MockStorage) {
        storage.HookReadHeader(func(hash types.Hash) (*types.Header, error) {
            return emptyHeader, nil
        })
    }

    blockchain, err := NewMockBlockchain(map[TestCallbackType]interface{}{
        StorageCallback: storageCallback,
    })
    if err != nil {
        t.Fatalf("unable to instantiate new blockchain, %v", err)
    }

    // Create a dummy block with a number much higher than the parent
    block := &types.Block{
        Header: &types.Header{
            Number: 10,
        },
    }

    assert.ErrorIs(t, blockchain.verifyBlockParent(block), ErrParentHashMismatch)
})

t.Run("Invalid block sequence", func(t *testing.T) {
    t.Parallel()

    // Set up the storage callback
    storageCallback := func(storage *storage.MockStorage) {
        storage.HookReadHeader(func(hash types.Hash) (*types.Header, error) {
            return emptyHeader, nil
        })
    }

```

```

blockchain, err := NewMockBlockchain(map[TestCallbackType]interface{}{
    StorageCallback: storageCallback,
})
if err != nil {
    t.Fatalf("unable to instantiate new blockchain, %v", err)
}

// Create a dummy block with a number much higher than the parent
block := &types.Block{
    Header: &types.Header{
        Number:      10,
        ParentHash: emptyHeader.Hash,
    },
}

assert.ErrorIs(t, blockchain.verifyBlockParent(block), ErrInvalidBlockSequence)
})

t.Run("Invalid block gas limit", func(t *testing.T) {
    t.Parallel()

    parentHeader := emptyHeader.Copy()
    parentHeader.GasLimit = 5000

    // Set up the storage callback
    storageCallback := func(storage *storage.MockStorage) {
        storage.HookReadHeader(func(hash types.Hash) (*types.Header, error) {
            return emptyHeader, nil
        })
    }

    blockchain, err := NewMockBlockchain(map[TestCallbackType]interface{}{
        StorageCallback: storageCallback,
    })
    if err != nil {
        t.Fatalf("unable to instantiate new blockchain, %v", err)
    }

    // Create a dummy block with a number much higher than the parent
    block := &types.Block{
        Header: &types.Header{
            Number:      1,
            ParentHash: parentHeader.Hash,
            GasLimit:    parentHeader.GasLimit + 1000, // The gas limit is greater than

```

the allowed rate

```
    },
}

    assert.Error(t, blockchain.verifyBlockParent(block))
})
}

// TestBlockchain_VerifyBlockBody makes sure that the block body is verified correctly
func TestBlockchain_VerifyBlockBody(t *testing.T) {
    t.Parallel()

    emptyHeader := &types.Header{
        Hash:      types.ZeroHash,
        ParentHash: types.ZeroHash,
    }

    t.Run("Invalid SHA3 Uncles root", func(t *testing.T) {
        t.Parallel()

        blockchain, err := NewMockBlockchain(nil)
        if err != nil {
            t.Fatalf("unable to instantiate new blockchain, %v", err)
        }

        block := &types.Block{
            Header: &types.Header{
                Sha3Uncles: types.ZeroHash,
            },
        }

        _, err = blockchain.verifyBlockBody(block)
        assert.ErrorIs(t, err, ErrInvalidSha3Uncles)
    })

    t.Run("Invalid Transactions root", func(t *testing.T) {
        t.Parallel()

        blockchain, err := NewMockBlockchain(nil)
        if err != nil {
            t.Fatalf("unable to instantiate new blockchain, %v", err)
        }

        block := &types.Block{
            Header: &types.Header{
```

```

        Sha3Uncles: types.EmptyUncleHash,
    },
}

_, err = blockchain.verifyBlockBody(block)
assert.ErrorIs(t, err, ErrInvalidTxRoot)
})

t.Run("Invalid execution result - missing parent", func(t *testing.T) {
    t.Parallel()

    // Set up the storage callback
    storageCallback := func(storage *storage.MockStorage) {
        storage.HookReadHeader(func(hash types.Hash) (*types.Header, error) {
            return nil, errors.New("not found")
        })
    }

    blockchain, err := NewMockBlockchain(map[TestCallbackType]interface{}{
        StorageCallback: storageCallback,
    })
    if err != nil {
        t.Fatalf("unable to instantiate new blockchain, %v", err)
    }

    block := &types.Block{
        Header: &types.Header{
            Sha3Uncles: types.EmptyUncleHash,
            TxRoot:     types.EmptyRootHash,
        },
    }

    _, err = blockchain.verifyBlockBody(block)
    assert.ErrorIs(t, err, ErrParentNotFound)
})

t.Run("Invalid execution result - unable to fetch block creator", func(t *testing.T) {
    t.Parallel()

    errBlockCreatorNotFound := errors.New("not found")

    // Set up the storage callback
    storageCallback := func(storage *storage.MockStorage) {
        // This is used for parent fetching
        storage.HookReadHeader(func(hash types.Hash) (*types.Header, error) {

```

```

        return emptyHeader, nil
    })
}

// Set up the verifier callback
verifierCallback := func(verifier *MockVerifier) {
    // This is used for error-ing out on the block creator fetch
    verifier.HookGetBlockCreator(func(t *types.Header) (types.Address, error) {
        return types.ZeroAddress, errBlockCreatorNotFound
    })
}

blockchain, err := NewMockBlockchain(map[TestCallbackType]interface{}{
    StorageCallback: storageCallback,
    VerifierCallback: verifierCallback,
})
if err != nil {
    t.Fatalf("unable to instantiate new blockchain, %v", err)
}

block := &types.Block{
    Header: &types.Header{
        Sha3Uncles: types.EmptyUncleHash,
        TxRoot:     types.EmptyRootHash,
    },
}

_, err = blockchain.verifyBlockBody(block)
assert.ErrorIs(t, err, errBlockCreatorNotFound)
})

t.Run("Invalid execution result - unable to execute transactions", func(t *testing.T) {
    t.Parallel()

    errUnableToExecute := errors.New("unable to execute transactions")

    // Set up the storage callback
    storageCallback := func(storage *storage.MockStorage) {
        // This is used for parent fetching
        storage.HookReadHeader(func(hash types.Hash) (*types.Header, error) {
            return emptyHeader, nil
        })
    }

    executorCallback := func(executor *mockExecutor) {

```

```

        // This is executor processing
        executor.HookProcessBlock(func(
            hash types.Hash,
            block *types.Block,
            address types.Address,
        ) (*state.Transition, error) {
            return nil, errUnableToExecute
        })
    }

    blockchain, err := NewMockBlockchain(map[TestCallbackType]interface{}{
        StorageCallback: storageCallback,
        ExecutorCallback: executorCallback,
    })
    if err != nil {
        t.Fatalf("unable to instantiate new blockchain, %v", err)
    }

    block := &types.Block{
        Header: &types.Header{
            Sha3Uncles: types.EmptyUncleHash,
            TxRoot:     types.EmptyRootHash,
        },
    }

    _, err = blockchain.verifyBlockBody(block)
    assert.ErrorIs(t, err, errUnableToExecute)
})

}

func TestBlockchain_CalculateBaseFee(t *testing.T) {
    t.Parallel()

    tests := []struct {
        blockNumber      uint64
        parentBaseFee     uint64
        parentGasLimit    uint64
        parentGasUsed     uint64
        expectedBaseFee   uint64
        elasticityMultiplier uint64
    }{
        {6, chain.GenesisBaseFee, 20000000, 10000000, chain.GenesisBaseFee, 2}, // usage ==
target
        {6, chain.GenesisBaseFee, 20000000, 10000000, 112500000, 4},           // usage ==
target
    }
}

```

```

        {6, chain.GenesisBaseFee, 20000000, 9000000, 987500000, 2}, // usage
below target
        {6, chain.GenesisBaseFee, 20000000, 9000000, 1100000000, 4}, // usage
below target
        {6, chain.GenesisBaseFee, 20000000, 11000000, 1012500000, 2}, // usage
above target
        {6, chain.GenesisBaseFee, 20000000, 11000000, 1150000000, 4}, // usage
above target
        {6, chain.GenesisBaseFee, 20000000, 20000000, 1125000000, 2}, // usage
full
        {6, chain.GenesisBaseFee, 20000000, 20000000, 1375000000, 4}, // usage
full
        {6, chain.GenesisBaseFee, 20000000, 0, 875000000, 2}, // usage 0
        {6, chain.GenesisBaseFee, 20000000, 0, 875000000, 4}, // usage 0
    }

    for i, test := range tests {
        test := test

        t.Run(fmt.Sprintf("%d", i), func(t *testing.T) {
            t.Parallel()

            blockchain := Blockchain{
                config: &chain.Chain{
                    Params: &chain.Params{
                        Forks: &chain.Forks{
                            chain.London: chain.NewFork(5),
                        },
                    },
                    Genesis: &chain.Genesis{
                        BaseFeeEM: test.elasticityMultiplier,
                    },
                },
            }

            parent := &types.Header{
                Number:    test.blockNumber,
                GasLimit:   test.parentGasLimit,
                GasUsed:    test.parentGasUsed,
                BaseFee:    test.parentBaseFee,
            }

            got := blockchain.CalculateBaseFee(parent)
            assert.Equal(t, test.expectedBaseFee, got, fmt.Sprintf("expected %d, got %d",
test.expectedBaseFee, got))

```



```

    })
}

func TestBlockchain_WriteFullBlock(t *testing.T) {
    t.Parallel()

    getKey := func(p []byte, k []byte) []byte {
        return append(append(make([]byte, 0, len(p)+len(k)), p...), k...)
    }
    db := map[string][]byte{}
    consensusMock := &MockVerifier{
        processHeadersFn: func(hs []*types.Header) error {
            assert.Len(t, hs, 1)

            return nil
        },
    }

    storageMock := storage.NewMockStorage()
    storageMock.HookNewBatch(func() storage.Batch {
        return memory.NewBatchMemory(db)
    })

    bc := &Blockchain{
        gpAverage: &gasPriceAverage{
            count: new(big.Int),
        },
        logger:    hclog.NewNullLogger(),
        db:        storageMock,
        consensus: consensusMock,
        config: &chain.Chain{
            Params: &chain.Params{
                Forks: &chain.Forks{
                    chain.London: chain.NewFork(5),
                },
            },
            Genesis: &chain.Genesis{
                BaseFeeEM: 4,
            },
        },
        stream: &eventStream{},
    }

    bc.headersCache, _ = lru.New(10)

```

```

bc.difficultyCache, _ = lru.New(10)

existingTD := big.NewInt(1)
existingHeader := &types.Header{Number: 1}
header := &types.Header{
    Number: 2,
}
receipts := []*types.Receipt{
    {GasUsed: 100},
    {GasUsed: 200},
}
tx := &types.Transaction{
    Value: big.NewInt(1),
}

tx.ComputeHash(1)
header.ComputeHash()
existingHeader.ComputeHash()
bc.currentHeader.Store(existingHeader)
bc.currentDifficulty.Store(existingTD)

header.ParentHash = existingHeader.Hash
bc.txSigner = &mockSigner{
    txFromByTxHash: map[types.Hash]types.Address{
        tx.Hash: {1, 2},
    },
}

// already existing block write
err := bc.WriteFullBlock(&types.FullBlock{
    Block: &types.Block{
        Header:      existingHeader,
        Transactions: []*types.Transaction{tx},
    },
    Receipts: receipts,
}, "polybft")

require.NoError(t, err)
require.Equal(t, 0, len(db))
require.Equal(t, uint64(1), bc.currentHeader.Load().Number)

// already existing block write
err = bc.WriteFullBlock(&types.FullBlock{
    Block: &types.Block{
        Header:      header,

```

```

        Transactions: []*types.Transaction{tx},
    },
    Receipts: receipts,
}, "polybft")

require.NoError(t, err)
require.Equal(t, 8, len(db))
require.Equal(t, uint64(2), bc.currentHeader.Load().Number)
require.NotNil(t, db[hex.EncodeToHex(getKey(storage.BODY, header.Hash.Bytes()))])
require.NotNil(t, db[hex.EncodeToHex(getKey(storage.TX_LOOKUP_PREFIX,
tx.Hash.Bytes()))])
require.NotNil(t, db[hex.EncodeToHex(getKey(storage.HEADER, header.Hash.Bytes()))])
require.NotNil(t, db[hex.EncodeToHex(getKey(storage.HEAD, storage.HASH))])
require.NotNil(t, db[hex.EncodeToHex(getKey(storage.CANONICAL,
common.EncodeUint64ToBytes(header.Number)))]])
require.NotNil(t, db[hex.EncodeToHex(getKey(storage.DIFFICULTY, header.Hash.Bytes()))])
require.NotNil(t, db[hex.EncodeToHex(getKey(storage.CANONICAL,
common.EncodeUint64ToBytes(header.Number)))]])
require.NotNil(t, db[hex.EncodeToHex(getKey(storage.RECEIPTS, header.Hash.Bytes()))])
}

func TestAddBlock_InvalidParent(t *testing.T) {
    // Setup: create a new test blockchain (using your helper or direct constructor)
    bc := NewTestBlockchain(t, nil)

    // Create a block with a parent hash that does not exist in the chain
    invalidParentHash := types.Hash{0xAA, 0xBB, 0xCC}
    block := &types.Block{
        Header: &types.Header{
            ParentHash: invalidParentHash,
            Number:     1,
            // Fill other required fields as needed
        },
        // Fill in Transactions, Uncles, etc. as needed
    }
    block.Header.ComputeHash()

    // Act: try to write the block
    err := bc.WriteBlock(block, "test")

    // Assert: should return an error about parent not found
    require.Error(t, err)
    require.Contains(t, err.Error(), "parent block not found")
}

```

```

func TestGenesisBlockInitialization(t *testing.T) {
    bc := NewTestBlockchain(t, nil)

    // The genesis block should be present after initialization
    header := bc.Header()
    require.NotNil(t, header)
    require.Equal(t, uint64(0), header.Number)
}

func TestGetBlockByHash_NonExistent(t *testing.T) {
    bc := NewTestBlockchain(t, nil)

    // Try to retrieve a block with a random hash
    randomHash := types.Hash{0xDE, 0xAD, 0xBE, 0xEF}
    block, found := bc.GetBlockByHash(randomHash, true)
    require.False(t, found)
    require.Nil(t, block)
}

```

IBFT:

```

package ibft
import (
    "testing"
    "time"

    "github.com/stretchr/testify/assert"
    "github.com/stretchr/testify/require"
    "github.com/0xPolygon/polygon-edge/consensus/proto"
    "github.com/0xPolygon/polygon-edge/consensus/messages"
)

// TestIBFTBackend_CalculateHeaderTimestamp verifies that the header timestamp
// is successfully calculated
func TestIBFTBackend_CalculateHeaderTimestamp(t *testing.T) {
    t.Parallel()

    // Reference time
    now := time.Unix(time.Now().UTC().Unix(), 0) // Round down

    testTable := []struct {
        name          string
        parentTimestamp int64
    }

```

```

    currentTime    time.Time
    blockTime      uint64

    expectedTimestamp time.Time
}{
    {
        "Valid clock block timestamp",
        now.Add(time.Duration(-1) * time.Second).Unix(), // 1s before
        now,
        1,
        now, // 1s after
    },
    {
        "Next multiple block clock",
        now.Add(time.Duration(-4) * time.Second).Unix(), // 4s before
        now,
        3,
        roundUpTime(now, 3*time.Second),
    },
}

for _, testCase := range testTable {
    testCase := testCase

    t.Run(testCase.name, func(t *testing.T) {
        t.Parallel()

        i := &backendIBFT{
            blockTime: time.Duration(testCase.blockTime) * time.Second,
        }

        assert.Equal(
            t,
            testCase.expectedTimestamp.Unix(),
            i.calcHeaderTimestamp(
                uint64(testCase.parentTimestamp),
                testCase.currentTime,
            ).Unix(),
        )
    })
}

// TestIBFTBackend_RoundUpTime verifies time is rounded up correctly
func TestIBFTBackend_RoundUpTime(t *testing.T) {

```

```

t.Parallel()

// Reference time
now := time.Now().UTC()

calcExpected := func(time int64, multiple int64) int64 {
    if time%multiple == 0 {
        return time + multiple
    }

    return ((time + multiple/2) / multiple) * multiple
}

testTable := []struct {
    name      string
    time      time.Time
    multiple  time.Duration

    expectedTime int64
}{
    {
        "No rounding needed",
        now,
        0 * time.Second,
        now.Unix(),
    },
    {
        "Rounded up time even",
        now,
        2 * time.Second,
        calcExpected(now.Unix(), 2),
    },
}

for _, testCase := range testTable {
    testCase := testCase

    t.Run(testCase.name, func(t *testing.T) {
        t.Parallel()

        assert.Equal(
            t,
            testCase.expectedTime,
            roundUpTime(testCase.time, testCase.multiple).Unix(),
        )
    })
}

```

```

    })
}

func TestBuildProposal_InvalidParent(t *testing.T) {
    ibft := newTestBackendIBFT() // You need to implement or use an existing test helper
    view := &proto.View{Height: 100}
    // Simulate missing parent block
    proposal := ibft.BuildProposal(view)
    require.Nil(t, proposal)
}

func TestInsertProposal_InvalidProposal(t *testing.T) {
    ibft := newTestBackendIBFT()
    proposal := &proto.Proposal{RawProposal: []byte("invalid")}
    committedSeals := []*messages.CommittedSeal{}
    // Should not panic or write block
    ibft.InsertProposal(proposal, committedSeals)
    // Optionally, assert logs or state changes
}

```

Executor:

```

package state

import (
    "fmt"
    "math/big"
    "testing"

    "github.com/stretchr/testify/assert"
    "github.com/stretchr/testify/require"

    "github.com/PrivixAI-labs/Privix-node/chain"
    "github.com/PrivixAI-labs/Privix-node/state/runtime"
    "github.com/PrivixAI-labs/Privix-node/types"
)

func TestOverride(t *testing.T) {
    t.Parallel()

    state := newStateWithPreState(map[types.Address]*PreState{
        {0x0}: {

```

```

        Nonce: 1,
        Balance: 1,
        State: map[types.Hash]types.Hash{
            types.ZeroHash: {0x1},
        },
    },
    {0x1}: {
        State: map[types.Hash]types.Hash{
            types.ZeroHash: {0x1},
        },
    },
})

nonce := uint64(2)
balance := big.NewInt(2)
code := []byte{0x1}

tt := NewTransition(chain.ForksInTime{}, state, newTxn(state))

require.Empty(t, tt.state.GetCode(types.ZeroAddress))

err := tt.WithStateOverride(types.StateOverride{
    {0x0}: types.OverrideAccount{
        Nonce: &nonce,
        Balance: balance,
        Code: code,
        StateDiff: map[types.Hash]types.Hash{
            types.ZeroHash: {0x2},
        },
    },
    {0x1}: types.OverrideAccount{
        State: map[types.Hash]types.Hash{
            {0x1}: {0x1},
        },
    },
})
require.NoError(t, err)

require.Equal(t, nonce, tt.state.GetNonce(types.ZeroAddress))
require.Equal(t, balance, tt.state.GetBalance(types.ZeroAddress))
require.Equal(t, code, tt.state.GetCode(types.ZeroAddress))
require.Equal(t, types.Hash{0x2}, tt.state.GetState(types.ZeroAddress, types.ZeroHash))

// state is fully replaced
require.Equal(t, types.Hash{0x0}, tt.state.GetState(types.Address{0x1},

```



```

types.ZeroHash))
    require.Equal(t, types.Hash{0x1}, tt.state.GetState(types.Address{0x1},
types.Hash{0x1}))
}

func Test_Transition_checkDynamicFees(t *testing.T) {
    t.Parallel()

    tests := []struct {
        name      string
        baseFee   *big.Int
        tx         *types.Transaction
        wantErr    assert.ErrorAssertionFunc
    }{
        {
            name:      "happy path",
            baseFee:   big.NewInt(100),
            tx: &types.Transaction{
                Type:      types.DynamicFeeTx,
                GasFeeCap: big.NewInt(100),
                GasTipCap: big.NewInt(100),
            },
            wantErr: func(t assert.TestingT, err error, i ...interface{}) bool {
                assert.NoError(t, err, i)

                return false
            },
        },
        {
            name:      "happy path with empty values",
            baseFee:   big.NewInt(0),
            tx: &types.Transaction{
                Type:      types.DynamicFeeTx,
                GasFeeCap: big.NewInt(0),
                GasTipCap: big.NewInt(0),
            },
            wantErr: func(t assert.TestingT, err error, i ...interface{}) bool {
                assert.NoError(t, err, i)

                return false
            },
        },
        {
            name:      "gas fee cap less than base fee",
            baseFee:   big.NewInt(20),

```

```

    tx: &types.Transaction{
        Type:      types.DynamicFeeTx,
        GasFeeCap: big.NewInt(10),
        GasTipCap:  big.NewInt(0),
    },
    wantErr: func(t assert.TestingT, err error, i ...interface{}) bool {
        expectedError := fmt.Sprintf("max fee per gas less than block base fee: "+
            "address %s, GasFeeCap: 10, BaseFee: 20", types.ZeroAddress)
        assert.EqualError(t, err, expectedError, i)

        return true
    },
},
{
    name:      "gas fee cap less than tip cap",
    baseFee: big.NewInt(5),
    tx: &types.Transaction{
        Type:      types.DynamicFeeTx,
        GasFeeCap: big.NewInt(10),
        GasTipCap:  big.NewInt(15),
    },
    wantErr: func(t assert.TestingT, err error, i ...interface{}) bool {
        expectedError := fmt.Sprintf("max priority fee per gas higher than max fee
per gas: "+
            "address %s, GasTipCap: 15, GasFeeCap: 10", types.ZeroAddress)
        assert.EqualError(t, err, expectedError, i)

        return true
    },
},
}

for _, tt := range tests {
    tt := tt

    t.Run(tt.name, func(t *testing.T) {
        t.Parallel()

        tr := &Transition{
            ctx: runtime.TxContext{
                BaseFee: tt.baseFee,
            },
        }

        err := tr.checkDynamicFees(tt.tx)

```

```

        tt.wantErr(t, err, fmt.Sprintf("checkDynamicFees(%v)", tt.tx))
    })
}

func TestProcessBlock_InvalidTransaction(t *testing.T) {
    exec := NewTestExecutor() // Use your test helper or construct with mocks
    parentRoot := types.ZeroHash
    block := &types.Block{
        Header: &types.Header{GasLimit: 1000000},
        Transactions: []*types.Transaction{
            {Gas: 2000000}, // Exceeds block gas limit
        },
    }
    _, err := exec.ProcessBlock(parentRoot, block, types.ZeroAddress)
    require.Error(t, err)
}

func TestWriteGenesis_InvalidStateRoot(t *testing.T) {
    exec := NewTestExecutor()
    alloc := map[types.Address]*chain.GenesisAccount{}
    // Use a non-existent state root
    _, err := exec.WriteGenesis(alloc, types.Hash{0xFF})
    require.Error(t, err)
}

```

Secrets:

```

package secrets

import (
    "testing"

    "github.com/stretchr/testify/assert"
    "github.com/stretchr/testify/require"
)

type dummySecretsManager struct {
    secrets map[string][]byte
}

func (d *dummySecretsManager) Setup() error { return nil }
func (d *dummySecretsManager) GetSecret(name string) ([]byte, error) {
    v, ok := d.secrets[name]

```

```

    if !ok {
        return nil, ErrSecretNotFound
    }
    return v, nil
}

func (d *dummySecretsManager) SetSecret(name string, value []byte) error {
    d.secrets[name] = value
    return nil
}

func (d *dummySecretsManager) HasSecret(name string) bool {
    _, ok := d.secrets[name]
    return ok
}

func (d *dummySecretsManager) RemoveSecret(name string) error {
    delete(d.secrets, name)
    return nil
}

func TestSupportedServiceManager(t *testing.T) {
    testTable := []struct {
        name          string
        serviceName SecretsManagerType
        supported      bool
    }{
        {
            "Valid local secrets manager",
            Local,
            true,
        },
        {
            "Valid Hashicorp Vault secrets manager",
            HashicorpVault,
            true,
        },
        {
            "Valid AWS SSM secrets manager",
            AWSSSM,
            true,
        },
        {
            "Valid GCP secrets manager",
            GCPSSM,
            true,
        },
    }
}

```

```

        "Invalid secrets manager",
        "MarsSecretsManager",
        false,
    },
}

for _, testCase := range testTable {
    t.Run(testCase.name, func(t *testing.T) {
        assert.Equal(
            t,
            testCase.supported,
            SupportedServiceManager(testCase.serviceName),
        )
    })
}
}

func TestSecretsManager_Basic(t *testing.T) {
    mgr := &dummySecretsManager{secrets: make(map[string][]byte)}
    err := mgr.SetSecret(ValidatorKey, []byte("keydata"))
    require.NoError(t, err)
    v, err := mgr.GetSecret(ValidatorKey)
    require.NoError(t, err)
    require.Equal(t, []byte("keydata"), v)
    require.True(t, mgr.HasSecret(ValidatorKey))
    require.NoError(t, mgr.RemoveSecret(ValidatorKey))
    require.False(t, mgr.HasSecret(ValidatorKey))
}

```

Server side:

```

package network

import (
    "context"
    "fmt"
    "net"
    "strconv"
    "testing"
    "time"

    "github.com/PrivixAI-labs/Privix-node/network/common"
    peerEvent "github.com/PrivixAI-labs/Privix-node/network/event"

```

```

"github.com/PrivixAI-labs/Privix-node/helper/tests"

"github.com/libp2p/go-libp2p/core/crypto"
"github.com/libp2p/go-libp2p/core/network"
"github.com/libp2p/go-libp2p/core/peer"
"github.com/multiformats/go-multiaddr"
"github.com/stretchr/testify/assert"
"github.com/hashicorp/go-hclog"
"github.com/stretchr/testify/require"
)

func TestConnLimit_Inbound(t *testing.T) {
    // we should not receive more inbound connections if we are already connected to max
    peers
    defaultConfig := &CreateServerParams{
        ConfigCallback: func(c *Config) {
            c.MaxInboundPeers = 1
            c.MaxOutboundPeers = 1
            c.NoDiscover = true
        },
    }

    servers, createErr := createServers(3, map[int]*CreateServerParams{
        0: defaultConfig,
        1: defaultConfig,
        2: defaultConfig,
    })
    if createErr != nil {
        t.Fatalf("Unable to create servers, %v", createErr)
    }

    t.Cleanup(func() {
        closeTestServers(t, servers)
    })

    // One slot left, Server 0 can connect to Server 1
    if joinErr := JoinAndWait(servers[0], servers[1], DefaultBufferTimeout,
DefaultJoinTimeout); joinErr != nil {
        t.Fatalf("Unable to join servers, %v", joinErr)
    }

    // Server 2 tries to connect to Server 1
    // but Server 1 is already connected to max inbound peers
    smallTimeout := time.Second * 5
    if joinErr := JoinAndWait(servers[2], servers[1], smallTimeout, smallTimeout); joinErr

```

```

== nil {
    t.Fatal("Peer join should've failed", joinErr)
}

// Disconnect Server 0 from Server 1 so Server 1 will have free slots
servers[0].DisconnectFromPeer(servers[1].host.ID(), "bye")

    disconnectCtx, disconnectFn := context.WithTimeout(context.Background(),
DefaultJoinTimeout)
    defer disconnectFn()

    if _, disconnectErr := WaitUntilPeerDisconnectsFrom(
        disconnectCtx,
        servers[0],
        servers[1].AddrInfo().ID,
    ); disconnectErr != nil {
        t.Fatalf("Unable to disconnect from peer, %v", disconnectErr)
    }

    // Attempt a connection between Server 2 and Server 1 again
    if joinErr := JoinAndWait(servers[2], servers[1], DefaultBufferTimeout,
DefaultJoinTimeout); joinErr != nil {
        t.Fatalf("Unable to join servers, %v", joinErr)
    }
}

func TestConnLimit_Outbound(t *testing.T) {
    // we should not try to make connections if we are already connected to max peers
    defaultConfig := &CreateServerParams{
        ConfigCallback: func(c *Config) {
            c.MaxInboundPeers = 1
            c.MaxOutboundPeers = 1
            c.NoDiscover = true
        },
    }

    servers, createErr := createServers(3, map[int]*CreateServerParams{
        0: defaultConfig,
        1: defaultConfig,
        2: defaultConfig,
    })
    if createErr != nil {
        t.Fatalf("Unable to create servers, %v", createErr)
    }
}

```

```

t.Cleanup(func() {
    closeTestServers(t, servers)
})

// One slot left, Server 0 can connect to Server 1
if joinErr := JoinAndWait(servers[0], servers[1], DefaultBufferTimeout,
DefaultJoinTimeout); joinErr != nil {
    t.Fatalf("Unable to join servers, %v", joinErr)
}

// Attempt to connect Server 0 to Server 2, but it should fail since
// Server 0 already has 1 peer (Server 1)
smallTimeout := time.Second * 5
if joinErr := JoinAndWait(servers[0], servers[2], smallTimeout, smallTimeout); joinErr
== nil {
    t.Fatalf("Unable to join servers, %v", joinErr)
}

// Disconnect Server 0 from Server 1
servers[0].DisconnectFromPeer(servers[1].host.ID(), "bye")

disconnectCtx, disconnectFn := context.WithTimeout(context.Background(),
DefaultJoinTimeout)
defer disconnectFn()

if _, disconnectErr := WaitUntilPeerDisconnectsFrom(
    disconnectCtx,
    servers[0],
    servers[1].AddrInfo().ID,
); disconnectErr != nil {
    t.Fatalf("Unable to wait for disconnect from peer, %v", disconnectErr)
}

// Now Server 0 is trying to connect to Server 2 since there are slots left
waitCtx, cancelWait := context.WithTimeout(context.Background(), DefaultJoinTimeout*2)
defer cancelWait()

_, err := WaitUntilPeerConnectsTo(waitCtx, servers[0], servers[2].host.ID())
if err != nil {
    t.Fatalf("Unable to wait for peer connect, %v", err)
}
}

func TestPeerEvent_EmitAndSubscribe(t *testing.T) {
    server, createErr := CreateServer(&CreateServerParams{ConfigCallback: func(c *Config) {

```



```

        c.NoDiscover = true
    })
    if createErr != nil {
        t.Fatalf("Unable to create server, %v", createErr)
    }

    t.Cleanup(func() {
        assert.NoError(t, server.Close())
    })

    receiver := make(chan *peerEvent.PeerEvent)

    err := server.Subscribe(context.Background(), func(evt *peerEvent.PeerEvent) {
        receiver <- evt
    })
    assert.NoError(t, err)

    count := 10
    events := []peerEvent.PeerEventType{
        peerEvent.PeerConnected,
        peerEvent.PeerFailedToConnect,
        peerEvent.PeerDisconnected,
        peerEvent.PeerDialCompleted,
        peerEvent.PeerAddedToDialQueue,
    }

    getIDAndEventType := func(i int) (peer.ID, peerEvent.PeerEventType) {
        id := peer.ID(strconv.Itoa(i))
        event := events[i%len(events)]

        return id, event
    }

    t.Run("Serial event emit and read", func(t *testing.T) {
        for i := 0; i < count; i++ {
            id, event := getIDAndEventType(i)
            server.EmitEvent(id, event)

            received := <-receiver
            assert.Equal(t, &peerEvent.PeerEvent{
                PeerID: id,
                Type:    event,
            }, received)
        }
    })

```

```

t.Run("Async event emit and read", func(t *testing.T) {
    for i := 0; i < count; i++ {
        id, event := getIDAndEventType(i)
        server.emitEvent(id, event)
    }
    for i := 0; i < count; i++ {
        received := <-receiver
        id, event := getIDAndEventType(i)
        assert.Equal(t, &peerEvent.PeerEvent{
            PeerID: id,
            Type:    event,
        }, received)
    }
})
}

func TestEncodingPeerAddr(t *testing.T) {
    _, pub, err := crypto.GenerateKeyPair(crypto.Secp256k1, 256)
    assert.NoError(t, err)

    id, err := peer.IDFromPublicKey(pub)
    assert.NoError(t, err)

    addr, err := multiaddr.NewMultiaddr("/ip4/127.0.0.1/tcp/90")
    assert.NoError(t, err)

    info := &peer.AddrInfo{
        ID:    id,
        Addrs: []multiaddr.Multiaddr{addr},
    }

    str, err := common.AddrInfoToString(info)
    assert.NoError(t, err)

    info2, err := common.StringToAddrInfo(str)
    assert.NoError(t, err)
    assert.Equal(t, info, info2)
}

func TestAddrInfoToString(t *testing.T) {
    defaultPeerID := peer.ID("123")
    defaultPort := 5000

    // formatMultiaddr is a helper function for formatting an IP / port combination

```

```

formatMultiaddr := func(ipVersion string, address string, port int) string {
    return fmt.Sprintf("/%s/%s/tcp/%d", ipVersion, address, port)
}

// formatExpectedOutput is a helper function for generating the expected output for
AddrInfoToString
formatExpectedOutput := func(input string, peerID string) string {
    return fmt.Sprintf("%s/p2p/%s", input, peerID)
}

testTable := []struct {
    name      string
    addresses []string
    expected  string
}{
    {
        // A host address is explicitly specified
        "Valid dial address found",
        []string{formatMultiaddr("ip4", "192.168.1.1", defaultPort)},
        formatExpectedOutput(
            formatMultiaddr("ip4", "192.168.1.1", defaultPort),
            defaultPeerID.String(),
        ),
    },
    {
        // A dns name is explicitly specified
        "Valid dial dns address",
        []string{formatMultiaddr("dns", "foobar.com", defaultPort)},
        formatExpectedOutput(
            formatMultiaddr("dns", "foobar.com", defaultPort),
            defaultPeerID.String(),
        ),
    },
    {
        // Multiple addresses that the host can listen on (0.0.0.0 special case)
        "Ignore IPv4 loopback address",
        []string{
            formatMultiaddr("ip4", "127.0.0.1", defaultPort),
            formatMultiaddr("ip4", "192.168.1.1", defaultPort),
        },
        formatExpectedOutput(
            formatMultiaddr("ip4", "192.168.1.1", defaultPort),
            defaultPeerID.String(),
        ),
    },
}

```

```

    {
        // Multiple addresses that the host can listen on (0.0.0.0 special case)
        "Ignore IPv6 loopback addresses",
        []string{
            formatMultiaddr("ip6", "0:0:0:0:0:0:0:1", defaultPort),
            formatMultiaddr("ip6", "::1", defaultPort),
            formatMultiaddr("ip6", "2001:0db8:85a3:0000:0000:8a2e:0370:7334",
defaultPort),
        },
        formatExpectedOutput(
            formatMultiaddr("ip6", "2001:db8:85a3::8a2e:370:7334", defaultPort),
            defaultPeerID.String(),
        ),
    },
}

for _, testCase := range testTable {
    t.Run(testCase.name, func(t *testing.T) {
        // Construct the multiaddrs
        multiAddrs, constructErr := constructMultiAddrs(testCase.addresses)
        if constructErr != nil {
            t.Fatalf("Unable to construct multiaddrs, %v", constructErr)
        }

        dialAddress, err := common.AddrInfoToString(&peer.AddrInfo{
            ID:    defaultPeerID,
            Addrs: multiAddrs,
        })

        assert.NoError(t, err)
        assert.Equal(t, testCase.expected, dialAddress)
    })
}

func TestJoinWhenAlreadyConnected(t *testing.T) {
    // if we try to join an already connected node, the watcher
    // should finish as well
    servers, createErr := createServers(2, nil)
    if createErr != nil {
        t.Fatalf("Unable to create servers, %v", createErr)
    }

    t.Cleanup(func() {
        closeTestServers(t, servers)
    })
}

```

```

    })

    // Server 0 should connect to Server 1
    if joinErr := JoinAndWait(servers[0], servers[1], DefaultBufferTimeout,
DefaultJoinTimeout); joinErr != nil {
        t.Fatalf("Unable to join servers, %v", joinErr)
    }

    // Server 1 should attempt to connect to Server 0, but shouldn't error out
    // if since it's already connected
    if joinErr := JoinAndWait(servers[1], servers[0], DefaultBufferTimeout,
DefaultJoinTimeout); joinErr != nil {
        t.Fatalf("Unable to join servers, %v", joinErr)
    }
}

func TestNat(t *testing.T) {
    testIP := "192.0.2.1"
    testPort := 1500 // important to be less than 2000 because of other tests and more than
1024 because of OS security
    testMultiAddrString := fmt.Sprintf("/ip4/%s/tcp/%d", testIP, testPort)

    server, createErr := CreateServer(&CreateServerParams{ConfigCallback: func(c *Config) {
        c.NatAddr = net.ParseIP(testIP)
        c.Addr.Port = testPort
    }})
    if createErr != nil {
        t.Fatalf("Unable to create server, %v", createErr)
    }

    t.Cleanup(func() {
        assert.NoError(t, server.Close())
    })

    // There should be multiple listening addresses
    listenAddresses := server.host.Network().ListenAddresses()
    assert.Greater(t, len(listenAddresses), 1)

    // NAT IP should not be found in listen addresses
    for _, addr := range listenAddresses {
        assert.NotEqual(t, addr.String(), testMultiAddrString)
    }

    // There should only be a singly registered server address
    addrInfo := server.AddrInfo()

```

```

    registeredAddresses := addrInfo.Addrs

    assert.Equal(t, 1, len(registeredAddresses))

    // NAT IP should be found in registered server addresses
    found := false

    for _, addr := range registeredAddresses {
        if addr.String() == testMultiAddrString {
            found = true

            break
        }
    }

    assert.True(t, found)
}

// TestPeerReconnection checks whether the node is able to reconnect with bootnodes on
// losing all active connections
func TestPeerReconnection(t *testing.T) {
    bootnodeConfig := &CreateServerParams{
        ConfigCallback: func(c *Config) {
            c.MaxInboundPeers = 3
            c.MaxOutboundPeers = 3
            c.NoDiscover = false
        },
    }

    // Create bootnodes
    bootnodes, createErr := createServers(2, map[int]*CreateServerParams{0: bootnodeConfig,
1: bootnodeConfig})
    if createErr != nil {
        t.Fatalf("Unable to create servers, %v", createErr)
    }

    t.Cleanup(func() {
        closeTestServers(t, bootnodes)
    })

    defaultConfig := &CreateServerParams{
        ConfigCallback: func(c *Config) {
            c.MaxInboundPeers = 3
            c.MaxOutboundPeers = 3
            c.NoDiscover = false

```

```

    },
    ServerCallback: func(server *Server) {
        addr1, err := common.AddrInfoToString(bootnodes[0].AddrInfo())
        assert.NoError(t, err)

        addr2, err := common.AddrInfoToString(bootnodes[1].AddrInfo())
        assert.NoError(t, err)

        server.config.Chain.Bootnodes = []string{addr1, addr2}
    },
}

servers, createErr := createServers(2, map[int]*CreateServerParams{0: defaultConfig, 1:
defaultConfig})
if createErr != nil {
    t.Fatalf("Unable to create servers, %v", createErr)
}

t.Cleanup(func() {
    for indx, server := range servers {
        if indx != 1 {
            // servers[1] is closed within the test
            assert.NoError(t, server.Close())
        }
    }
})

disconnectFromPeer := func(server *Server, peerID peer.ID) {
    server.DisconnectFromPeer(peerID, "Bye")

    disconnectCtx, disconnectFn := context.WithTimeout(context.Background(),
DefaultJoinTimeout)
    defer disconnectFn()

    if _, disconnectErr := WaitUntilPeerDisconnectsFrom(disconnectCtx, server, peerID);
disconnectErr != nil {
        t.Fatalf("Unable to wait for disconnect from peer, %v", disconnectErr)
    }
}

closePeerServer := func(server *Server, peer *Server) {
    peerID := peer.AddrInfo().ID

    if closeErr := peer.Close(); closeErr != nil {
        t.Fatalf("Unable to close server, %v", closeErr)
    }
}

```

```

    }

    disconnectCtx, disconnectFn := context.WithTimeout(context.Background(),
DefaultJoinTimeout)
    defer disconnectFn()

    if _, disconnectErr := WaitUntilPeerDisconnectsFrom(disconnectCtx, server, peerID);
disconnectErr != nil {
        t.Fatalf("Unable to wait for disconnect from peer, %v", disconnectErr)
    }
}

// connect with the first boot node
if joinErr := JoinAndWait(servers[0], bootnodes[0], DefaultBufferTimeout,
DefaultJoinTimeout); joinErr != nil {
    t.Fatalf("Unable to join servers, %v", joinErr)
}

// connect with the second boot node
if joinErr := JoinAndWait(servers[0], bootnodes[1], DefaultBufferTimeout,
DefaultJoinTimeout); joinErr != nil {
    t.Fatalf("Unable to join servers, %v", joinErr)
}

// Connect with the second server
if joinErr := JoinAndWait(servers[0], servers[1], DefaultBufferTimeout,
DefaultJoinTimeout); joinErr != nil {
    t.Fatalf("Unable to join servers, %v", joinErr)
}

// disconnect from the first boot node
disconnectFromPeer(servers[0], bootnodes[0].AddrInfo().ID)

// disconnect from the second boot node
disconnectFromPeer(servers[0], bootnodes[1].AddrInfo().ID)

// disconnect from the other server
closePeerServer(servers[0], servers[1])

waitCtx, cancelWait := context.WithTimeout(context.Background(), DefaultJoinTimeout*3)
defer cancelWait()

reconnected, err := WaitUntilPeerConnectsTo(waitCtx, servers[0],
bootnodes[0].host.ID(), bootnodes[1].host.ID())
if err != nil {

```



```

        t.Fatalf("Unable to wait for peer connect, %v", err)
    }

    assert.True(t, reconnected)
}

func TestReconnectionWithNewIP(t *testing.T) {
    natIP := "127.0.0.1"

    _, dir0 := GenerateTestLibp2pKey(t)
    _, dir1 := GenerateTestLibp2pKey(t)

    defaultConfig := func(c *Config) {
        c.NoDiscover = true
    }

    servers, createErr := createServers(3,
        map[int]*CreateServerParams{
            0: {
                ConfigCallback: func(c *Config) {
                    defaultConfig(c)
                    c.DataDir = dir0
                },
            },
            1: {
                ConfigCallback: func(c *Config) {
                    defaultConfig(c)
                    c.DataDir = dir1
                },
            },
            2: {
                ConfigCallback: func(c *Config) {
                    defaultConfig(c)
                    c.DataDir = dir1
                    // same ID to but different IP from servers[1]
                    c.NatAddr = net.ParseIP(natIP)
                },
            },
        },
    )

    if createErr != nil {
        t.Fatalf("Unable to create servers, %v", createErr)
    }

    t.Cleanup(func() {

```

```

        closeTestServers(t, servers)
    })

    // Server 0 should connect to Server 1
    if joinErr := JoinAndWait(servers[0], servers[1], DefaultBufferTimeout,
DefaultJoinTimeout); joinErr != nil {
        t.Fatalf("Unable to join servers, %v", joinErr)
    }

    // Server 1 terminates, so Server 0 should disconnect from it
    peerID := servers[1].AddrInfo().ID

    if err := servers[1].host.Close(); err != nil {
        t.Fatalf("Unable to close peer server, %v", err)
    }

    disconnectCtx, disconnectFn := context.WithTimeout(context.Background(),
DefaultJoinTimeout)
    defer disconnectFn()

    if _, disconnectErr := WaitUntilPeerDisconnectsFrom(disconnectCtx, servers[0], peerID);
disconnectErr != nil {
        t.Fatalf("Unable to wait for disconnect from peer, %v", disconnectErr)
    }

    // servers[0] connects to servers[2]
    // Server 0 should connect to Server 2 (that has the NAT address set)
    if joinErr := JoinAndWait(servers[0], servers[2], DefaultBufferTimeout,
DefaultJoinTimeout); joinErr != nil {
        t.Fatalf("Unable to join servers, %v", joinErr)
    }

    // Wait until Server 2 also has a connection to Server 0 before asserting
    connectCtx, connectFn := context.WithTimeout(context.Background(),
DefaultBufferTimeout)
    defer connectFn()

    if _, connectErr := WaitUntilPeerConnectsTo(
        connectCtx,
        servers[2],
        servers[0].AddrInfo().ID,
    ); connectErr != nil {
        t.Fatalf("Unable to wait for connection between Server 2 and Server 0, %v",
connectErr)
    }

```

```

    assert.Equal(t, int64(1), servers[0].numPeers())
    assert.Equal(t, int64(1), servers[2].numPeers())
}

func TestSelfConnection_WithBootNodes(t *testing.T) {
    // Create a temporary directory for storing the key file
    key, directoryName := GenerateTestLibp2pKey(t)
    peerID, err := peer.IDFromPrivateKey(key)
    assert.NoError(t, err)
    testMultiAddr := tests.GenerateTestMultiAddr(t).String()
    peerAddressInfo, err := common.StringToAddrInfo(testMultiAddr)
    assert.NoError(t, err)

    testTable := []struct {
        name          string
        bootNodes     []string
        expectedList  []*peer.AddrInfo
    }{
        {
            name:          "Should return an non empty bootnodes list",
            bootNodes:     []string{"/ip4/127.0.0.1/tcp/10001/p2p/" + peerID.Pretty(),
testMultiAddr},
            expectedList:  []*peer.AddrInfo{peerAddressInfo},
        },
    }

    for _, tt := range testTable {
        t.Run(tt.name, func(t *testing.T) {
            server, createErr := CreateServer(&CreateServerParams{
                ConfigCallback: func(c *Config) {
                    c.NoDiscover = false
                    c.DataDir = directoryName
                },
                ServerCallback: func(server *Server) {
                    server.config.Chain.Bootnodes = tt.bootNodes
                },
            })
            if createErr != nil {
                t.Fatalf("Unable to create server, %v", createErr)
            }

            assert.Equal(t, tt.expectedList, server.bootnodes.getBootnodes())
        })
    }
}

```

```

    }
}

func TestRunDial(t *testing.T) {
    t.Skip()
    // setupServers returns server and list of peer's server
    setupServers := func(t *testing.T, maxPeers []int64) []*Server {
        t.Helper()

        servers := make([]*Server, len(maxPeers))
        for idx := range servers {
            server, createErr := CreateServer(
                &CreateServerParams{
                    ConfigCallback: func(c *Config) {
                        c.MaxInboundPeers = maxPeers[idx]
                        c.MaxOutboundPeers = maxPeers[idx]
                        c.NoDiscover = true
                    },
                })
            if createErr != nil {
                t.Fatalf("Unable to create servers, %v", createErr)
            }

            servers[idx] = server
        }

        return servers
    }

    closeServers := func(servers ...*Server) {
        for _, s := range servers {
            assert.NoError(t, s.Close())
        }
    }

    t.Run("should connect to all peers", func(t *testing.T) {
        maxPeers := []int64{2, 1, 1}
        servers := setupServers(t, maxPeers)
        srv, peers := servers[0], servers[1:]

        for _, p := range peers {
            if joinErr := JoinAndWait(srv, p, DefaultBufferTimeout, DefaultJoinTimeout);
joinErr != nil {
                t.Fatalf("Unable to join peer, %v", joinErr)
            }
        }
    })
}

```

```

    }
    closeServers(servers...)
})

t.Run("should fail to connect to some peers due to reaching limit", func(t *testing.T) {
    maxPeers := []int64{2, 1, 1, 1}
    servers := setupServers(t, maxPeers)
    srv, peers := servers[0], servers[1:]

    for idx, p := range peers {
        if int64(idx) < maxPeers[0] {
            // Connection should be successful
            joinErr := JoinAndWait(srv, p, DefaultBufferTimeout, DefaultJoinTimeout)
            assert.NoError(t, joinErr)
        } else {
            // Connection should fail
            smallTimeout := time.Second * 5
            joinErr := JoinAndWait(srv, p, smallTimeout, smallTimeout)
            assert.Error(t, joinErr)
        }
    }
    closeServers(servers...)
})

t.Run("should try to connect after adding a peer to queue", func(t *testing.T) {
    maxPeers := []int64{1, 0, 1}
    servers := setupServers(t, maxPeers)
    srv, peers := servers[0], servers[1:]

    // Server 1 can't connect to any peers, so this join should fail
    smallTimeout := time.Second * 5
    if joinErr := JoinAndWait(srv, peers[0], smallTimeout, smallTimeout); joinErr ==
nil {
        t.Fatalf("Shouldn't be able to join peer, %v", joinErr)
    }

    // Server 0 and Server 2 should connect
    if joinErr := JoinAndWait(srv, peers[1], DefaultBufferTimeout, DefaultJoinTimeout);
joinErr != nil {
        t.Fatalf("Couldn't join peer, %v", joinErr)
    }

    closeServers(srv, peers[1])
})

```

```

}

func TestSubscribe(t *testing.T) {
    t.Parallel()

    setupServer := func(t *testing.T, shouldCloseAfterTest bool) *Server {
        t.Helper()

        srv, err := CreateServer(
            &CreateServerParams{
                ConfigCallback: func(c *Config) {
                    c.MaxInboundPeers = 0
                    c.MaxOutboundPeers = 0
                    c.NoDiscover = true
                },
            },
        )

        if err != nil {
            t.Fatalf("Unable to create server, %v", err)
        }

        if shouldCloseAfterTest {
            t.Cleanup(func() {
                srv.Close()
            })
        }

        return srv
    }

    toChannel := func(t *testing.T, ctx context.Context, server *Server) <-chan
    *peerEvent.PeerEvent {
        t.Helper()

        eventCh := make(chan *peerEvent.PeerEvent)

        t.Cleanup(func() {
            close(eventCh)
        })

        err := server.Subscribe(ctx, func(e *peerEvent.PeerEvent) {
            eventCh <- e
        })
    }
}

```

```

    assert.NoError(t, err)

    return eventCh
}

waitForEvent := func(
    t *testing.T,
    eventCh <-chan *peerEvent.PeerEvent,
    timeout time.Duration,
) (*peerEvent.PeerEvent, bool) {
    t.Helper()

    select {
    case received := <-eventCh:
        return received, true
    case <-time.After(timeout):
        return nil, false
    }
}

event := &peerEvent.PeerEvent{
    PeerID: peer.ID("test"),
    Type:   peerEvent.PeerConnected,
}

t.Run("should call callback", func(t *testing.T) {
    t.Parallel()

    server := setupServer(t, true)

    ctx, cancel := context.WithCancel(context.Background())
    t.Cleanup(func() {
        cancel()
    })

    eventCh := toChannel(t, ctx, server)

    server.EmitEvent(event)

    res, received := waitForEvent(t, eventCh, time.Second*5)

    assert.True(t, received)
    assert.Equal(t, event, res)
})

```

```

t.Run("should not call callback after context is closed", func(t *testing.T) {
    t.Parallel()

    server := setupServer(t, true)

    ctx, cancel := context.WithCancel(context.Background())

    eventCh := toChannel(t, ctx, server)

    // cancel before emitting
    cancel()

    server.EmitEvent(event)

    _, received := waitForEvent(t, eventCh, time.Second*5)

    assert.False(t, received)
})

```

```

t.Run("should not call callback after server closed", func(t *testing.T) {
    t.Parallel()

    server := setupServer(t, false)

    ctx, cancel := context.WithCancel(context.Background())
    t.Cleanup(func() {
        cancel()
    })

    eventCh := toChannel(t, ctx, server)

    // close server before emitting event
    server.Close()

    server.EmitEvent(event)

    _, received := waitForEvent(t, eventCh, time.Second*5)

    assert.False(t, received)
})

```

```

}

```

```

func TestMinimumBootNodeCount(t *testing.T) {
    tests := []struct {
        name      string

```



```

    bootNodes    []string
    expectedError error
}{
    {
        name:      "Server config with no bootnodes",
        bootNodes: nil,
        expectedError: ErrNoBootnodes,
    },
    {
        name:      "Server config with less than one bootnode",
        bootNodes: []string{},
        expectedError: ErrMinBootnodes,
    },
    {
        name:      "Server config with at least one bootnode",
        bootNodes: []string{tests.GenerateTestMultiAddr(t).String()},
        expectedError: nil,
    },
}

for _, tt := range tests {
    t.Run(tt.name, func(t *testing.T) {
        _, createErr := CreateServer(&CreateServerParams{
            ServerCallback: func(server *Server) {
                server.config.Chain.Bootnodes = tt.bootNodes
            },
        })

        if tt.expectedError != nil {
            assert.ErrorAs(t, tt.expectedError, &createErr)
        } else {
            assert.NoError(t, createErr)
        }
    })
}
}

func TestMultiAddrFromDns(t *testing.T) {
    port := 12345

    tests := []struct {
        name      string
        dnsAddress string
        port      int
        err       bool
    }

```

```

outcome    string
}{
{
    name:        "Invalid DNS Version",
    dnsAddress:  "/dns8/example.io/",
    port:        port,
    err:         true,
    outcome:     "",
},
{
    name:        "Invalid DNS String",
    dnsAddress:  "dns4rahul.io",
    port:        port,
    err:         true,
    outcome:     "",
},
{
    name:        "Valid DNS Address with `/'` ",
    dnsAddress:  "/dns4/rahul.io/",
    port:        port,
    err:         false,
    outcome:     fmt.Sprintf("/dns4/rahul.io/tcp/%d", port),
},
{
    name:        "Valid DNS Address without `/'` ",
    dnsAddress:  "dns6/example.io",
    port:        port,
    err:         false,
    outcome:     fmt.Sprintf("/dns6/example.io/tcp/%d", port),
},
{
    name:        "Invalid Port Number",
    dnsAddress:  "dns6/example.io",
    port:        100000,
    err:         true,
    outcome:     "",
},
{
    name:        "Invalid Host name starting with `-'` ",
    dnsAddress:  "dns6/-example.io",
    port:        port,
    err:         true,
    outcome:     "",
},
{

```

```

        name:      "Invalid Host name starting with `/\` ",
        dnsAddress: "dns6//example.io",
        port:      port,
        err:       true,
        outcome:    "",
    },
    {
        name:      "Invalid Host name  with `/\` ",
        dnsAddress: "dns6/example/.io",
        port:      12345,
        err:       true,
        outcome:    "",
    },
    {
        name:      "Invalid Host name  with `-` ",
        dnsAddress: "dns6/example-.io",
        port:      port,
        err:       true,
        outcome:    "",
    },
    {
        name:      "Missing DNS version",
        dnsAddress: "example.io",
        port:      port,
        err:       true,
        outcome:    "",
    },
    {
        name:      "Invalid DNS version",
        dnsAddress: "/dns8/example.io",
        port:      port,
        err:       true,
        outcome:    "",
    },
    {
        name:      "valid long domain suffix",
        dnsAddress: "dns/validator-1.foo.technology",
        port:      port,
        err:       false,
        outcome:    fmt.Sprintf("/dns/validator-1.foo.technology/tcp/%d", port),
    },
}

for _, tt := range tests {
    t.Run(tt.name, func(t *testing.T) {

```

```

        multiAddr, err := common.MultiAddrFromDNS(tt.dnsAddress, tt.port)
        if !tt.err {
            assert.NotNil(t, multiAddr, "Multi Address should not be nil")
            assert.Equal(t, multiAddr.String(), tt.outcome)
        } else {
            assert.Error(t, err)
        }
    })
}
}

```

*// TestPeerAdditionDeletion tests that the server's peer connection
 // information handling is valid*

```

func TestPeerAdditionDeletion(t *testing.T) {
    createServer := func() *Server {
        server, createErr := CreateServer(nil)
        if createErr != nil {
            t.Fatalf("Unable to create networking server, %v", createErr)
        }

        return server
    }

    generateAndAddPeers := func(server *Server, peersNum int) []*randomPeer {
        randomPeers, err := generateRandomPeers(t, peersNum)
        if err != nil {
            t.Fatalf("Unable to generate random peers, %v", err)
        }

        for _, randomPeer := range randomPeers {
            server.AddPeer(randomPeer.peerID, randomPeer.direction)

            assert.True(t, true, server.hasPeer(randomPeer.peerID))
        }

        assert.Len(t, server.Peers(), peersNum)

        return randomPeers
    }

    extractExpectedDirectionCounts := func(randomPeers []*randomPeer) (
        expectedOutbound int64,
        expectedInbound int64,
    ) {
        for _, randPeer := range randomPeers {

```

```

        if randPeer.direction == network.DirOutbound {
            expectedOutbound++

            continue
        }

        expectedInbound++
    }

    return
}

validateConnectionCounts := func(
    server *Server,
    expectedOutbound int64,
    expectedInbound int64,
) {
    assert.Equal(t, expectedOutbound, server.connectionCounts.GetOutboundConnCount())
    assert.Equal(t, expectedInbound, server.connectionCounts.GetInboundConnCount())
}

t.Run("peers are added correctly", func(t *testing.T) {
    generateAndAddPeers(createServer(), 2500)
})

t.Run("no duplicate peers added", func(t *testing.T) {
    server := createServer()

    randomPeers, err := generateRandomPeers(t, 1)
    if err != nil {
        t.Fatalf("Unable to generate random peers, %v", err)
    }

    randomPeer := randomPeers[0]

    server.AddPeer(randomPeer.peerID, randomPeer.direction)

    assert.True(t, true, server.hasPeer(randomPeer.peerID))

    server.AddPeer(randomPeer.peerID, randomPeer.direction)

    assert.Len(t, server.Peers(), 1)

    outbound, inbound := extractExpectedDirectionCounts(randomPeers)
    validateConnectionCounts(server, outbound, inbound)

```

```

}))

t.Run("existing peer with the opposite conn. direction", func(t *testing.T) {
    server := createServer()

    randomPeers, err := generateRandomPeers(t, 1)
    if err != nil {
        t.Fatalf("Unable to generate random peers, %v", err)
    }

    randPeer := randomPeers[0]

    newDirection := network.DirInbound
    if newDirection == randPeer.direction {
        newDirection = network.DirOutbound
    }

    randomPeerOppositeDirection := &randomPeer{
        peerID:    randPeer.peerID,
        direction: newDirection,
    }

    // Add all peer variations to the server
    randomPeers = append(randomPeers, randomPeerOppositeDirection)
    for _, peer := range randomPeers {
        server.AddPeer(peer.peerID, peer.direction)

        assert.True(t, true, server.hasPeer(peer.peerID))
    }

    assert.Len(t, server.Peers(), 1)

    // Make sure the directions match
    for indx, connInfo := range server.Peers() {
        assert.Equal(t, randomPeers[indx].peerID, connInfo.Info.ID)
        assert.True(t, connInfo.connDirections[network.DirOutbound])
        assert.True(t, connInfo.connDirections[network.DirInbound])
    }

    outbound, inbound := extractExpectedDirectionCounts(randomPeers)
    validateConnectionCounts(server, outbound, inbound)
}))

t.Run("peers are removed correctly", func(t *testing.T) {
    server := createServer()

```

```

    peersNum := 10

    // Generate and add the random peers
    randomPeers := generateAndAddPeers(server, peersNum)

    // Prune off every other peer
    prunedPeers := 0
    for i := 0; i < len(randomPeers); i += 2 {
        prunedPeers++
        server.removePeer(randomPeers[i].peerID)

        assert.False(t, server.hasPeer(randomPeers[i].peerID))
    }

    leftoverPeers := make([]*randomPeer, 0)
    for i := 1; i < len(randomPeers); i += 2 {
        leftoverPeers = append(leftoverPeers, randomPeers[i])
    }

    // Make sure the peers lists match
    assert.Len(t, server.Peers(), peersNum-prunedPeers)

    outbound, inbound := extractExpectedDirectionCounts(leftoverPeers)
    validateConnectionCounts(server, outbound, inbound)
}))
}

type randomPeer struct {
    peerID    peer.ID
    direction network.Direction
}

// generateRandomPeers generates random peer data
func generateRandomPeers(t *testing.T, count int) ([]*randomPeer, error) {
    t.Helper()

    randomPeers := make([]*randomPeer, count)

    for i := 0; i < count; i++ {
        testMultiAddr := tests.GenerateTestMultiAddr(t).String()

        peerAddressInfo, err := common.StringToAddrInfo(testMultiAddr)
        if err != nil {
            return nil, err
        }
    }
}

```

```

        // Get a random direction
        randDirection := network.DirOutbound
        if i%2 == 0 {
            randDirection = network.DirInbound
        }

        randomPeers[i] = &randomPeer{
            peerID:      peerAddressInfo.ID,
            direction: randDirection,
        }
    }

    return randomPeers, nil
}

func TestNewServer_InvalidKey(t *testing.T) {
    config := &Config{ /* set up with invalid secrets manager */ }
    _, err := NewServer(hclog.NewNullLogger(), config)
    require.Error(t, err)
}

func TestServer_PeerConnection(t *testing.T) {
    // This would require a mock or in-memory libp2p host
    // Example: create two servers, connect, check peer count
}

```

Crypto:

```

package crypto

import (
    "bytes"
    "math/big"
    "os"
    "strconv"
    "strings"
    "testing"
    "time"

    "github.com/PrivixAI-labs/Privix-node/helper/hex"
    "github.com/PrivixAI-labs/Privix-node/types"
    "github.com/stretchr/testify/assert"
    "github.com/stretchr/testify/require"

```



```

)

func TestKeyEncoding(t *testing.T) {
    for i := 0; i < 10; i++ {
        priv, _ := GenerateECDSAKey()

        // marshall private key
        buf, err := MarshalECDSAPrivateKey(priv)
        assert.NoError(t, err)

        priv0, err := ParseECDSAPrivateKey(buf)
        assert.NoError(t, err)

        assert.Equal(t, priv, priv0)

        // marshall public key
        buf = MarshalPublicKey(&priv.PublicKey)

        pub0, err := ParsePublicKey(buf)
        assert.NoError(t, err)

        assert.Equal(t, priv.PublicKey, *pub0)
    }
}

func TestCreate2(t *testing.T) {
    t.Parallel()

    cases := []struct {
        address string
        salt     string
        initCode string
        result   string
    }{
        {
            "0x0000000000000000000000000000000000000000",
            "0x0000000000000000000000000000000000000000000000000000000000000000",
            "0x00",
            "0x4D1A2e2bB4F88F0250f26Ffff098B0b30B26BF38",
        },
        {
            "0xdeadbeef0000000000000000000000000000000000",
            "0x0000000000000000000000000000000000000000000000000000000000000000",
            "0x00",
            "0xB928f69Bb1D91Cd65274e3c79d8986362984fDA3",
        },
    }
}

```

```

    },
    {
        "0xdeadbeef00000000000000000000000000000000",
        "0x0000000000000000000000000000000000000000000000000000000000000000",
        "0x00",
        "0xD04116cDd17beBE565EB2422F2497E06cC1C9833",
    },
    {
        "0x000000000000000000000000000000000000000000000000000",
        "0x0000000000000000000000000000000000000000000000000000000000000000",
        "0xdeadbeef",
        "0x70f2b2914A2a4b783FaEFb75f459A580616Fcb5e",
    },
    {
        "0x0000000000000000000000000000000000000000000000000000000000000000",
        "0x0000000000000000000000000000000000000000000000000000000000000000",
        "0xdeadbeef",
        "0x60f3f640a8508fC6a86d45DF051962668E1e8AC7",
    },
    {
        "0x0000000000000000000000000000000000000000000000000000000000000000",
        "0x0000000000000000000000000000000000000000000000000000000000000000",
    },
    {
        "0xdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef",
        "0x1d8bFDc5D46DC4f61D6b6115972536eBE6A8854C",
    },
    {
        "0x0000000000000000000000000000000000000000000000000000000000000000",
        "0x0000000000000000000000000000000000000000000000000000000000000000",
        "0x",
        "0xE33C0C7F7df4809055C3ebA6c09CFe4BaF1BD9e0",
    },
}

for _, c := range cases {
    c := c

    t.Run("", func(t *testing.T) {
        t.Parallel()

        address := types.StringToAddress(c.address)
        initCode := hex.MustDecodeHex(c.initCode)

        saltRaw := hex.MustDecodeHex(c.salt)

```

```

        if len(saltRaw) != 32 {
            t.Fatal("Salt length must be 32 bytes")
        }

        salt := [32]byte{}
        copy(salt[:], saltRaw[:])

        res := CreateAddress2(address, salt, initCode)

        // values in the test cases are in EIP155 format, toLower until
        // the EIP155 is done.
        assert.Equal(t, strings.ToLower(c.result), strings.ToLower(res.String()))
    })
}

func TestValidateSignatureValues(t *testing.T) {
    t.Parallel()

    var (
        zero      = big.NewInt(0)
        one       = big.NewInt(1)
        two       = big.NewInt(2)
        minusOne  = big.NewInt(-1)

        limit      = secp256k1N
        limitMinus1 = new(big.Int).Sub(secp256k1N, one)
        halfLimit  = new(big.Int).Div(secp256k1N, two)
        doubleLimit = new(big.Int).Mul(secp256k1N, two)

        // smaller than secp256k1N but bytes.Compare returns it's bigger
        smallValue, _ = new(big.Int).SetString("ffffffffffffffffffffffffffffebadd", 16)
    )

    // make sure smallValue is less than secp256k1N by big.Int.Compare
    assert.Equal(
        t,
        limit.Cmp(smallValue),
        1,
        "small value must be less than secp256k1N",
    )

    // make sure smallValue is greater than secp256k1N by bytes.Compare
    assert.Equal(
        t,

```

```

bytes.Compare(smallValue.Bytes(), limit.Bytes()),
1,
"small value must be greater than secp256k1N by lexicographical comparison",
)

cases := []struct {
    homestead bool
    name       string
    v          *big.Int
    r          *big.Int
    s          *big.Int
    res        bool
}{
    // correct v, r, s
    {
        name:      "should be valid if v is 0 and r & s are in range",
        homestead: true, v: zero, r: one, s: one, res: true,
    },
    {
        name:      "should be valid if v is 1 and r & s are in range",
        homestead: true, v: one, r: one, s: one, res: true,
    },
    // incorrect v, correct r, s.
    {
        name:      "should be invalid if v is out of range",
        homestead: true, v: two, r: one, s: one, res: false,
    },
    {
        name:      "should be invalid if v is out of range",
        homestead: true, v: big.NewInt(-10), r: one, s: one, res: false,
    },
    {
        name:      "should be invalid if v is out of range",
        homestead: true, v: big.NewInt(10), r: one, s: one, res: false,
    },
    // incorrect v, incorrect/correct r, s.
    {
        name:      "should be invalid if v & r & s are out of range",
        homestead: true, v: two, r: zero, s: zero, res: false,
    },
    {
        name:      "should be invalid if v & r are out of range",
        homestead: true, v: two, r: zero, s: one, res: false,
    },
    {

```

```

    name:      "should be invalid if v & s are out of range",
    homestead: true, v: two, r: one, s: zero, res: false,
  },
  // correct v, incorrect r, s
  {
    name:      "should be invalid if r & s are nil",
    homestead: true, v: zero, r: nil, s: nil, res: false,
  },
  {
    name:      "should be invalid if r is nil",
    homestead: true, v: zero, r: nil, s: one, res: false,
  },
  {
    name:      "should be invalid if s is nil",
    homestead: true, v: zero, r: one, s: nil, res: false,
  },
  {
    name:      "should be invalid if r & s are negative",
    homestead: true, v: zero, r: minusOne, s: minusOne, res: false,
  },
  {
    name:      "should be invalid if r is negative",
    homestead: true, v: zero, r: minusOne, s: one, res: false,
  },
  {
    name:      "should be invalid if s is negative",
    homestead: true, v: zero, r: one, s: minusOne, res: false,
  },
  {
    name:      "should be invalid if r & s are out of range",
    homestead: true, v: zero, r: zero, s: zero, res: false,
  },
  {
    name:      "should be invalid if r is out of range (v = 0)",
    homestead: true, v: zero, r: zero, s: one, res: false,
  },
  {
    name:      "should be invalid if s is out of range (v = 0)",
    homestead: true, v: zero, r: one, s: zero, res: false,
  },
  {
    name:      "should be invalid if r & s are out of range (v = 1)",
    homestead: true, v: one, r: zero, s: zero, res: false,
  },
  {

```

```

    name:      "should be invalid if r is out of range (v = 1)",
    homestead: true, v: one, r: zero, s: one, res: false,
  },
  {
    name:      "should be invalid if s is out of range (v = 1)",
    homestead: true, v: one, r: one, s: zero, res: false,
  },
  // incorrect r, s max limit (Frontier)
  {
    name:      "should be invalid if r & s equal to secp256k1N in Frontier",
    homestead: false, v: zero, r: limit, s: limit, res: false,
  },
  {
    name:      "should be invalid if r equals to secp256k1N in Frontier",
    homestead: false, v: zero, r: limit, s: limitMinus1, res: false,
  },
  {
    name:      "should be invalid if s equals to secp256k1N in Frontier",
    homestead: false, v: zero, r: limitMinus1, s: limit, res: false,
  },
  // incorrect r, s max limit (Homestead)
  {
    name:      "should be invalid if r & s equal to secp256k1N in Homestead",
    homestead: true, v: zero, r: limit, s: limit, res: false,
  },
  {
    name:      "should be invalid if r equals to secp256k1N in Homestead",
    homestead: true, v: zero, r: limit, s: limitMinus1, res: false,
  },
  {
    name:      "should be invalid if s equals to secp256k1N in Homestead",
    homestead: true, v: zero, r: limitMinus1, s: limit, res: false,
  },
  // frontier v, r, s max limit (Frontier)
  {
    name:      "should be valid if r & s equal to secp256k1N - 1 in Frontier",
    homestead: false, v: zero, r: limitMinus1, s: limitMinus1, res: true,
  },
  // incorrect v, r, s max limit (Homestead)
  {
    name:      "should be invalid if r & s equal to secp256k1N - 1 in Homestead",
    homestead: true, v: zero, r: limitMinus1, s: limitMinus1, res: false,
  },
  // correct v, r, s max limit (Homestead)
  {

```

```

        name:      "should be valid if r equals to secp256k1N - 1 and s equals to
secp256k1N/2",
        homestead: true, v: zero, r: limitMinus1, s: halfLimit, res: true,
    },
    // edge cases
    // Previously ValidateSignatureValues uses bytes.Compare to compare r and s with
upper limit
    // but bytes.Compare compares them lexicographically then it causes strange
judgment
    // e.g.
    // 0xfffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141
    // > 0x01fffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141 (double
value)
    {
        name:      "should be invalid if r & s equal to 2 * secp256k1N in Frontier",
        homestead: false, v: zero, r: doubleLimit, s: doubleLimit, res: false,
    },
    {
        name:      "should be invalid if r equals to 2 * secp256k1N in Frontier",
        homestead: false, v: zero, r: doubleLimit, s: one, res: false,
    },
    {
        name:      "should be invalid if s equals to 2 * secp256k1N in Frontier",
        homestead: false, v: zero, r: one, s: doubleLimit, res: false,
    },
    {
        name:      "should be invalid if r equals to 2 * secp256k1N and s equal to
secp256k1N",
        homestead: true, v: zero, r: doubleLimit, s: limit, res: false,
    },
    {
        name:      "should be invalid if r equals to 2 * secp256k1N in Frontier",
        homestead: true, v: zero, r: doubleLimit, s: one, res: false,
    },
    {
        name:      "should be invalid if s equals to secp256k1N in Frontier",
        homestead: true, v: zero, r: one, s: limit, res: false,
    },
    {
        name:      "should be valid if r & s equal to small value",
        homestead: false, v: zero, r: smallValue, s: smallValue, res: true,
    },
    {
        name:      "should be valid if r equals to smallValue in Frontier",
        homestead: false, v: zero, r: smallValue, s: one, res: true,
    },

```

```

    },
    {
        name:      "should be valid if s equals to smallValue in Frontier",
        homestead: false, v: zero, r: one, s: smallValue, res: true,
    },
}

for _, c := range cases {
    c := c

    t.Run(c.name, func(t *testing.T) {
        t.Parallel()

        assert.Equal(
            t,
            c.res,
            ValidateSignatureValues(c.v, c.r, c.s, c.homestead),
        )
    })
}

}

func TestValidateSignatureValues_Invalid(t *testing.T) {
    v := big.NewInt(2)
    r := big.NewInt(0) // Invalid: should be > 0
    s := big.NewInt(1)
    isHomestead := true
    valid := ValidateSignatureValues(v, r, s, isHomestead)
    require.False(t, valid)
}

func TestSignAndVerifyECDSA(t *testing.T) {
    priv, _ := GenerateECDSAKey()
    msg := []byte("hello world")
    hash := Keccak256(msg)
    sig, err := Sign(priv, hash)
    require.NoError(t, err)
    pub, err := RecoverPubkey(sig, hash)
    require.NoError(t, err)
    require.NotNil(t, pub)
}

func TestPrivateKeyRead(t *testing.T) {
    t.Parallel()

```



```

// Write private keys to disk, check if read is ok
testTable := []struct {
    name           string
    privateKeyHex   string
    checksummedAddress string
    shouldFail      bool
}{
    // Generated with Ganache
    {
        "Valid address #1",
        "0c3d062cd3c642735af6a3c1492d761d39a668a67617a457113eaf50860e9e3f",
        "0x81e83Dc147B81Db5771D998A2C265cc710BE43a5",
        false,
    },
    {
        "Valid address #2",
        "71e6439122f6a44884132d54a978318d7218021a5d8f39fd24f440774d564d87",
        "0xCe1f32314aD63F18123b822a23c214DabAA9F7Cf",
        false,
    },
    {
        "Valid address #3",
        "c6435f6cb3a8f19111737b72944a0b4a7e52d8a6e95f1ebaa2881679f2087709",
        "0x47B7DAc4361062Dfc43d0EA6A2a4C3d27bBcCbdb",
        false,
    },
    {
        "Invalid key",
        "c6435f6cb3a8f19111737b72944a0b4a7e52d8a6e95f1ebaa2881679f",
        "",
        true,
    },
}

for _, testCase := range testTable {
    testCase := testCase
    t.Run(testCase.name, func(t *testing.T) {
        t.Parallel()

        privateKey, err := BytesToECDSAPrivateKey([]byte(testCase.privateKeyHex))
        if err != nil && !testCase.shouldFail {
            t.Fatalf("Unable to parse private key, %v", err)
        }

        if !testCase.shouldFail {

```

```

        address, err := GetAddressFromKey(privateKey)
        if err != nil {
            t.Fatalf("unable to extract key, %v", err)
        }

        assert.Equal(t, testCase.checksummedAddress, address.String())
    } else {
        assert.Nil(t, privateKey)
    }
})
}
}

func TestPrivateKeyGeneration(t *testing.T) {
    tempFile := "./privateKeyTesting-" + strconv.FormatInt(time.Now().UTC().Unix(), 10) +
        ".key"

    t.Cleanup(func() {
        _ = os.Remove(tempFile)
    })

    // Generate the private key and write it to a file
    writtenKey, err := GenerateOrReadPrivateKey(tempFile)
    if err != nil {
        t.Fatalf("Unable to generate private key, %v", err)
    }

    writtenAddress, err := GetAddressFromKey(writtenKey)
    if err != nil {
        t.Fatalf("unable to extract key, %v", err)
    }

    // Read existing key and check if it matches
    readKey, err := GenerateOrReadPrivateKey(tempFile)
    if err != nil {
        t.Fatalf("Unable to read private key, %v", err)
    }

    readAddress, err := GetAddressFromKey(readKey)
    if err != nil {
        t.Fatalf("unable to extract key, %v", err)
    }

    assert.True(t, writtenKey.Equal(readKey))
    assert.Equal(t, writtenAddress.String(), readAddress.String())
}

```

```

}

func TestRecoverPublicKey(t *testing.T) {
    t.Parallel()

    testSignature := []byte{1, 2, 3}

    t.Run("Empty hash", func(t *testing.T) {
        t.Parallel()

        _, err := RecoverPubkey(testSignature, []byte{})
        require.ErrorIs(t, err, errHashOfInvalidLength)
    })

    t.Run("Hash of non appropriate length", func(t *testing.T) {
        t.Parallel()

        _, err := RecoverPubkey(testSignature, []byte{0, 1})
        require.ErrorIs(t, err, errHashOfInvalidLength)
    })

    t.Run("Ok signature and hash", func(t *testing.T) {
        t.Parallel()

        hash := types.BytesToHash([]byte{0, 1, 2})

        privateKey, err := GenerateECDSAKey()
        require.NoError(t, err)

        signature, err := Sign(privateKey, hash.Bytes())
        require.NoError(t, err)

        publicKey, err := RecoverPubkey(signature, hash.Bytes())
        require.NoError(t, err)

        require.True(t, privateKey.PublicKey.Equal(publicKey))
    })
}

```

Technical Findings Summary

Findings

Vulnerability Level		Total	Pending	Not Apply	Acknowledged	Partially Fixed	Fixed
	HIGH	4					4
	MEDIUM	3					3
	LOW	0					
	INFORMATIONAL	1			1		

Assessment Results

Score Results

Review	Score
Global Score	90/100
Assure KYC	https://projects.assuredefi.com/project/privix
Audit Score	90/100

The Following Score System Has been Added to this page to help understand the value of the audit, the maximum score is 100, however to attain that value the project must pass and provide all the data needed for the assessment. Our Passing Score has been changed to 84 Points for a higher standard, if a project does not attain 85% is an automatic failure. Read our notes and final assessment below. The Global Score is a combination of the evaluations obtained between having or not having KYC and the type of contract audited together with its manual audit.

Audit PASS

The security audit did not pass. Multiple high-risk issues were identified across the network and supporting infrastructure. Notably;

1. Restore IBFT safety: revert proposer selection logic.
2. Lock down RPC: bind to loopback, require authentication, whitelist methods.
3. Harden P2P: throttle peers, validate message sizes, enable libp2p flood controls.
4. Use crypto/rand for all randomness.
5. Encrypt & lock down on-disk secrets; enforce strict FS permissions.
6. LevelDB: enable fsync, atomic writes, and hardened file perms.
7. CI pipeline: add vulnerability scanning (govulncheck) and dependency pinning.

By addressing the critical consensus flaws immediately and following the above mitigations, Privix-node can achieve robust safety, liveness, and operational security.

Update: The development team resolves all issues presented [commit - ea530da55d73586081edf3357b75406c292f06b2], Based on the fixes implemented and the team's response, **the audit is considered passed**. We recommend ongoing security after the deployment and continuous monitoring in mainnet.

Disclaimer

Assure Defi has conducted an independent security assessment to verify the integrity of and highlight any vulnerabilities or errors, intentional or unintentional, that may be present in the reviewed code for the scope of this assessment. This report does not constitute agreement, acceptance, or advocating for the Project, and users relying on this report should not consider this as having any merit for financial adMetabot in any shape, form, or nature. The contracts audited do not account for any economic developments that the Project in question may pursue, and the veracity of the findings thus presented in this report relate solely to the proficiency, competence, aptitude, and discretion of our independent auditors, who make no guarantees nor assurance that the contracts are entirely free of exploits, bugs, vulnerabilities or deprecation of technologies.

All information provided in this report does not constitute financial or investment adMetabot, nor should it be used to signal that any person reading this report should invest their funds without sufficient individual due diligence, regardless of the findings presented. Information is provided 'as is, and Assure Defi is under no covenant to audit completeness, accuracy, or solidity of the contracts. In no event will Assure Defi or its partners, employees, agents, or parties related to the provision of this audit report be liable to any parties for, or lack thereof, decisions or actions with regards to the information provided in this audit report.

The assessment serMetabots provided by Assure Defi are subject to dependencies and are under continuing development. You agree that your access or use, including but not limited to any serMetabots, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies with high levels of technical risk and uncertainty. The assessment reports could include false positives, negatives, and unpredictable results. The serMetabots may access, and depend upon, multiple layers of third parties.