BEOSIN
Blockchain Security

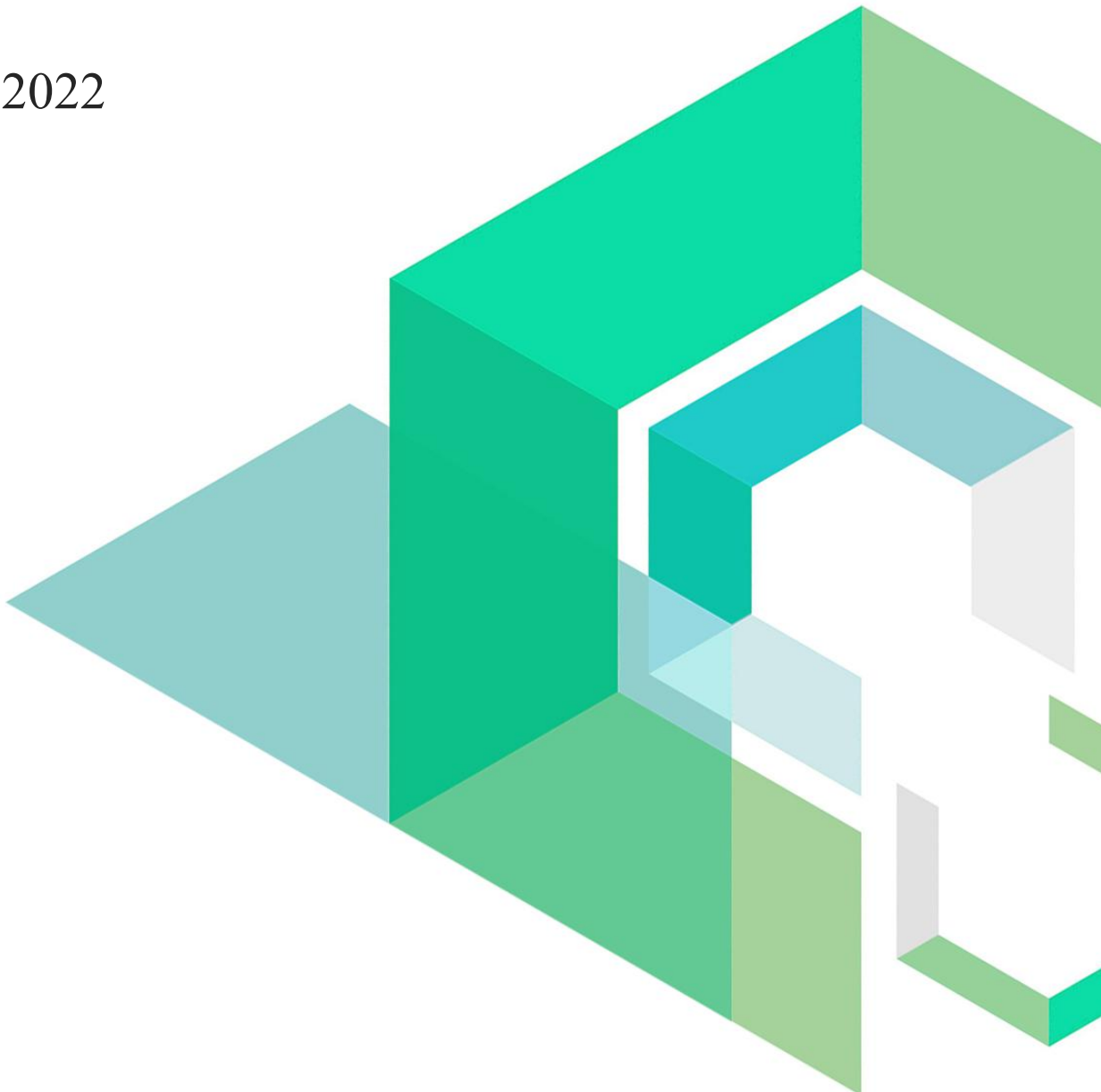
dogechain

Public Blockchain Security Audit

V1.0

No. 202206151650

Jun 15th, 2022



Contents

Overview **1**

Project Overview 1

Audit Overview 1

Summary of audit results **2**

Findings **3**

[dogechain-1] The transactions pool blockage 4

[dogechain-2] The --block-gas-target variable is not in effect 5

[dogechain-3] System contract management permission is too large 6

[dogechain-4] Possible voting problems in PoA consensus model 7

[dogechain-5] Inaccurate error message of *EstimateGas* interface 8

[dogechain-6] Exceptional error message in special case of eth_call 9

Audit Content **10**

1 JSON RPC Security Audit 10

1.1 JSON RPC Introduction 10

1.2 JSON RPC Processing Logic 11

1.3 RPC Sensitive Interface Permission 12

1.4 CLI commands security audit 12

2 Node Security 13

2.1 Number of Node Connections 13

2.2 Packet Size Limit 13

2.3 Node Network Access Restrictions 14

3 Account &Asset Security 15

3.1 Account Model 15

3.2 Account Generation..... 15
3.3 Transaction Signature..... 17
3.4 Asset Security..... 18
4 Consensus Security..... 19
4.1 IBFT-PoS Consensus Process Analysis..... 20
4.2 Validator Rotation Mechanism..... 21
4.3 Rewards for Building Blocks..... 22
4.4 Block Proposer Selection Algorithm Security Analysis..... 22
5 Transaction Model Security..... 23
5.1 Transaction Processing Flow..... 23
5.2 Replay Attack..... 25
5.3 Dusting Attack..... 26
5.4 Trading Flooding Attacks..... 27
5.5 Double-spending Attack..... 27
5.6 Illegal Transactions..... 27
5.7 Fake Deposit Attack..... 29
5.8 Transaction Pool Security Audit..... 29
6 System Contract Security..... 29
6.1 validatorSet Contract..... 30
6.2 bridge Contract..... 32
6.3 vault Contract..... 34
Disclaimer..... 35

Overview

Project Overview

Project Name	dogechain
Audit scope	https://github.com/dogechain-lab/dogechain/tree/v0.4.0 https://github.com/dogechain-lab/dogechain/tree/v0.4.1 https://github.com/dogechain-lab/dogechain/tree/v0.4.2 https://github.com/dogechain-lab/dogechain/tree/v0.5.0 https://github.com/dogechain-lab/dogechain/tree/v0.5.1
Commit Hash	v0.4.0: 4dfcea48a3948d435f4d03c742fffe173f979739 v0.4.1 4dfcea48a3948d435f4d03c742fffe173f979739 (Initial) 9dc14912dd7dcffe2bb58acc94b5a6d3cf80761c (Final) v0.4.2 4dfcea48a3948d435f4d03c742fffe173f979739 (Initial) 3c8a86f3ecff67b7690e10fbdb489e0bf81d52a5 (Final) v0.5.0: 4dfcea48a3948d435f4d03c742fffe173f979739 (Initial) 4c4593bd05f28b829c0797a6aa52d50e22b12a24 (Final) v0.5.1: f0ca5d38491f7b7c2919e0f1487f61980d478646 (Initial) 4b52db0f4d30ac6f8eb026c638bdd75c3fe6b249 (Final)

Audit Overview

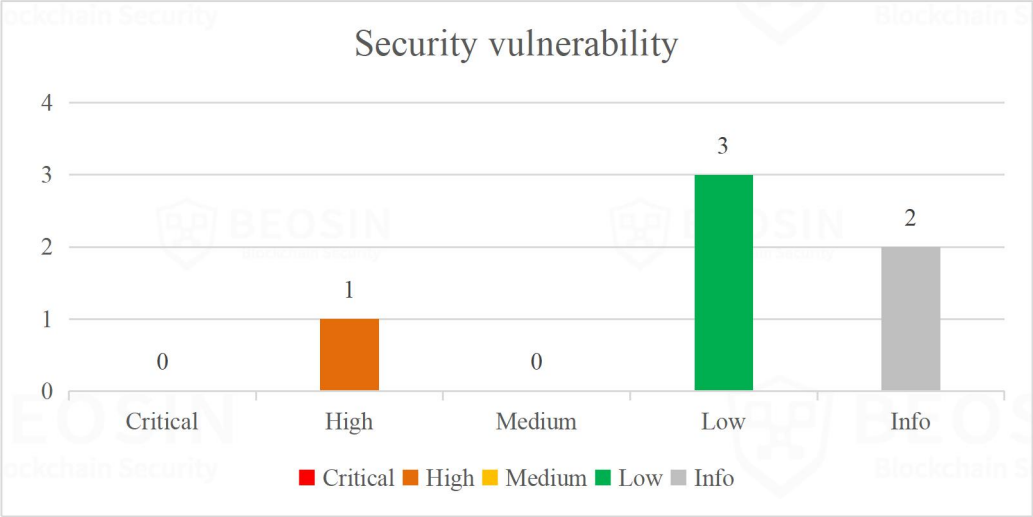
Audit work duration: Apr 21, 2022 – Jun 15, 2022

Audit methods: Formal Verification, Static Analysis, Typical Case Testing and Manual Review.

Audit team: Beosin Technology Co. Ltd.

Summary of audit results

After auditing, 1 High risks, 3 Low-risks and 2 Info items were identified in the Versailles-heroes project. Specific audit details will be presented in the **Findings section**. Users should pay attention to the following aspects when interacting with this project：



Notes：

- **Risk Description:**

1、Transactions with too high a nonce are left in the transaction pool and are not processed. If too many transactions are left in the transaction pool, the node's transaction pool will be blocked and cannot receive normal transactions.

2、If running with the IBFT-PoA consensus mechanism (the project part is committed to using IBFT-PoS), the voting function may be anomalous.

3、The dogechain system contract implements a high permission feature, so participating users should pay attention to the owner address of the system contract.

Findings

Index	Risk description	Severity level	Status
dogechain-1	The transactions pool blockage	High	Acknowledged
dogechain-2	The --block-gas-target variable is not in effect	Low	Fixed
dogechain-3	System contract management permission is too large	Low	Acknowledged
dogechain-4	Possible voting problems in PoA consensus model	Low	Acknowledged
dogechain-5	Inaccurate error message of *EstimateGas* interface	Info	Acknowledged
dogechain-6	Exceptional error message in special case of eth_call	Info	Acknowledged

[dogechain-1] The transactions pool blockage

Severity Level	**High**
Description	There is a max-slots limit for the node corresponding to the transaction pool in the dogechain. If the volume of transactions in the transaction pool reaches the max-slots, then the node will not accept new transactions. In this regard, if a transaction constructed with too high nonce is used and this part of the transaction is submitted to the specified node, then the waiting pool queued for that node will be filled up and it will not be able to process normal transactions.

```
jury.txpool: failed to add tx: err="txpool is full"
jury.dispatcher: failed to dispatch: method=eth_sendRawTransaction err="txpool is
```

Figure 1 Error message for transaction pool blocking

Recommendations	Add logic to clean up invalid transactions in the transaction pool.
Status	Acknowledged.

[dogechain-2] The --block-gas-target variable is not in effect

Severity Level	**Low**
Description	The value of the --block-gas-target variable can be specified to adjust the block's gas limit when the chain is started, but this variable is not written to the chain configuration when the chain is started, making it invalid.

```
func (p *serverParams) generateConfig() *server.Config {
    return &server.Config{
        Chain: p.genesisConfig,
        JSONRPC: &server.JSONRPC{
            JSONRPCAddr:              p.jsonRPCAddress,
            AccessControlAllowOrigin: p.corsAllowedOrigins,
        },
        GRPCAddr:   p.grpcAddress,
        LibP2PAddr: p.libp2pAddress,
        Telemetry: &server.Telemetry{
            PrometheusAddr: p.prometheusAddress,
        },
        Network: &network.Config{
            NoDiscover:       p.rawConfig.Network.NoDiscover,
            Addr:             p.libp2pAddress,
            NatAddr:          p.natAddress,
            DNS:              p.dnsAddress,
            DataDir:          p.rawConfig.DataDir,
            MaxPeers:         p.rawConfig.Network.MaxPeers,
            MaxInboundPeers:  p.rawConfig.Network.MaxInboundPeers,
            MaxOutboundPeers: p.rawConfig.Network.MaxOutboundPeers,
            Chain:            p.genesisConfig,
        },
        DataDir:             p.rawConfig.DataDir,
        Seal:                p.rawConfig.ShouldSeal,
        PriceLimit:          p.rawConfig.TxPool.PriceLimit,
        MaxSlots:            p.rawConfig.TxPool.MaxSlots,
        MaxAccountDemotions: p.rawConfig.TxPool.MaxAccountDemotions,
        SecretsManager:      p.secretsConfig,
        RestoreFile:         p.getRestoreFilePath(),
        BlockTime:           p.rawConfig.BlockTime,
        LogLevel:            hclog.LevelFromString(p.rawConfig.LogLevel),
        Daemon:              p.isDaemon,
        ValidatorKey:        p.validatorKey,
    }
}
```

Figure 2 Configuration parameter generation for chain start

Recommendations	Add logic to clean up invalid transactions in the transactions pool.
Status	Fixed. The issue has been fixed in 4c4593bd05f28b829c0797a6aa52d50e22b12a24 commit.

[dogechain-3] System contract management permission is too large

Severity Level	**Low**
Description	The current system contract validatorSet and bridge in the owner's permission is too high, such as: the owner can remove the pledge amount reaches the specified value of the address from the validatorSet; can manage the cross-chain signer and the number of signatures threshold, etc..
Recommendations	It is recommended to use a multi-signature wallet or DAO governance contract management system contract owner
Status	Acknowledged. The project party says it will use DAO for governance when it goes live.

[dogechain-4] Possible voting problems in PoA consensus model

Severity Level	**Low**
Description	In the IBFT-PoA mode, the validator needs to add and remove validators through validator voting, but if the validator has voted for the specified address, the validator cannot cancel the vote before the specified address becoming validator.

```
+/work/dogechain/0.4.1/jury$ jury ibft propose --grpc-address 127.0.0.1:10000 --addr 0x8B15464F8233F718c8605B16eBADA6fc09181fC2 -
code = Unknown desc = already a candidate
```

Figure 3 Remove the validator prompt message

Recommendations	Modify the check logic for removing validator votes.
Status	Acknowledged. After the feedback from the project, the dogechain will adopt the PoS model when it goes live and does not require a validator for voting.

[dogechain-5] Inaccurate error message of *EstimateGas* interface

Severity Level	Info
Description	As shown in the figure below, when calling the *EstimateGas* interface to budget gas, the *testTransaction* function will be called to pre-execute the transaction, then if the corresponding transaction does not fail due to gas (e.g., the contract function execution conditions are not met), the error message will also be given as: unable to apply transaction even for the highest gas limit.

```
// Check if the highEnd is a good value to make the transaction pass
failed, err := testTransaction(highEnd, false)
if failed {
    // The transaction shouldn't fail, for whatever reason, at highEnd
    return 0, fmt.Errorf(
        "unable to apply transaction even for the highest gas limit %d: %w",
        highEnd,
        err,
    )
}
```

Figure 4 Partial source code of *EstimateGas* function

It has been verified that the problem only occurs when calling the *EstimateGas* function and is considered by default to be caused by insufficient gas, which does not affect normal transactions.

Recommendations	It is recommended to modify the internal logic of *EstimateGas*.
Status	Acknowledged.

[dogechain-6] Exceptional error message in special case of eth_call

Severity Level	Info
Description	When calling the *eth_call* interface, if a query function is called (which should not consume gas), but incorrectly specifies a gas greater than the block-gas-limit, the query operation will fail with the following error message (non-query functions do not have this problem):

```
"error": {
    "code": -32600,
    "message": "gas limit reached in the pool"
}
```

Figure 5 Error message for exception call to *eth_call*

The reason for this is that the *apply* function is called in the *eth_call* function to check for gas and return the corresponding error message.

```
// 3. the amount of gas required is available in the block
if err := t.subGasPool(msg.Gas); err != nil {
    return nil, NewGasLimitReachedTransitionApplicationError(err)
}
```

Figure 6 Partial source code of *eth_call* function

Recommendations	Verify business requirements and choose whether to fix this question.
Status	Acknowledged.

Audit Content

The dogechain is a modular and extensible framework for building Ethereum-compatible block chain networks, sidechains, and general scaling solutions.

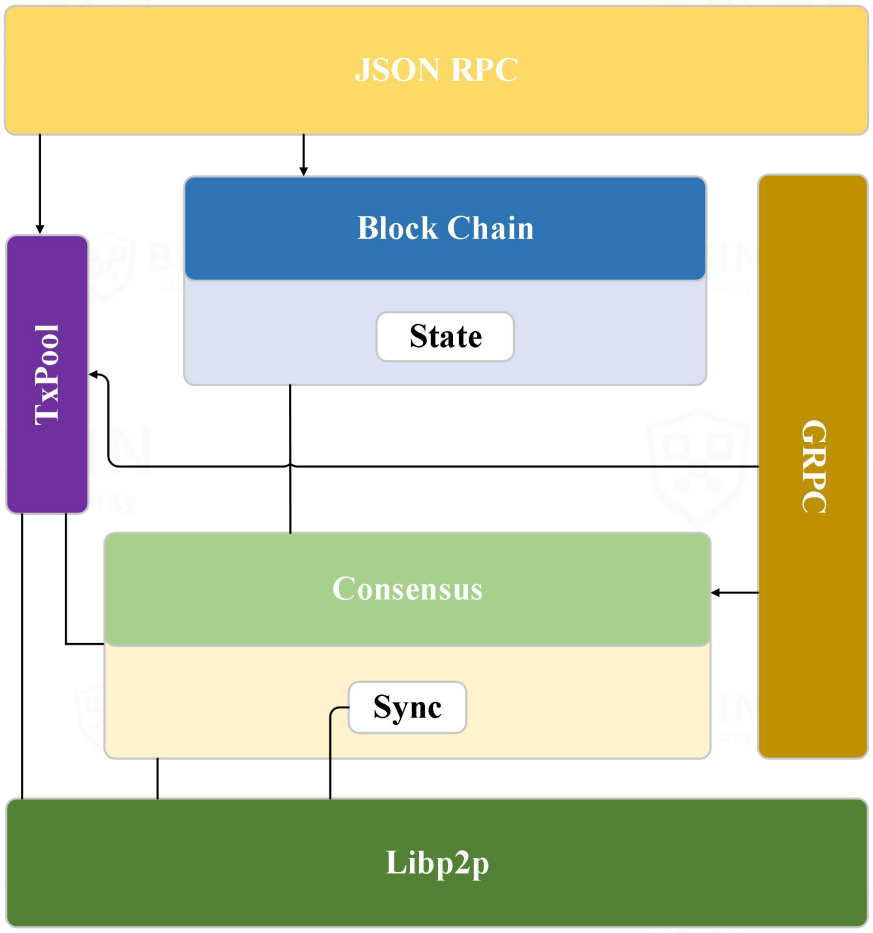


Figure 7 The dogechain overall framework

1 JSON RPC Security Audit

1.1 JSON RPC Introduction

RPC provides a way to access the exported objects through the grid or other I/O connections. RPC serializes the methods and parameters to be called and transmits them to the other side through TCP/UDP protocol, and the other side deserializes them to find the methods and parameters to be called, and then serializes the return values to return. After creating the RPC service, the object can be registered to the service to make it accessible to the outside. The export method according to this specific way is called remote call, which also supports the publish/subscribe model.

When the dogechain node starts, it will pull up the JSON RPC service, create dispatcher and filter manager, and register the corresponding RPC module at the same time. The JSON RPC implemented by dogechain currently contains four modules: eth, net, web3 and txpool.

```
d.endpoints.Eth = &Eth{d.logger, store, d.chainID, d.filterManager}
d.endpoints.Net = &Net{store, d.chainID}
d.endpoints.Web3 = &Web3{}
d.endpoints.TxPool = &TxPool{store}

d.registerService("eth", d.endpoints.Eth)
d.registerService("net", d.endpoints.Net)
d.registerService("web3", d.endpoints.Web3)
d.registerService("txpool", d.endpoints.TxPool)
```

Figure 8 JSON RPC module

The interfaces of net and web3 modules are basically the same as the corresponding modules of Ethereum, but dogechain will make some deletions in eth module compared with Ethereum, especially in transaction submission and signature. Currently dogechain does not directly provide *eth_sign* and *eth_sendTransaction* two core interfaces, but only supports *eth_sendRawTransaction* interface to send original transactions, which is related to the application scenario of dogechain itself.

The txpool module is a new RPC module of dogechain, which contains three interfaces: *txpool_content, txpool_inspect* and *txpool_status*. Among them, *txpool_content* will iterate through the transactions in the pending and queued lists of the current pool and return them; *txpool_inspect* will return the current and total capacity in addition to the transactions in the pending and queued lists; and the *txpool_status* will only return the number of transactions in the pending and queued lists. The status function only returns the number of transactions in the pending and queued lists.

1.2 JSON RPC Processing Logic

In dogechain, after receiving an RPC request, the node will first call the *handle* function to check the request function type; then to the *Handle* function on the Dispatcher to check the requested data body; finally, to parse the corresponding RPC request and locate the corresponding module and function to call.

```go
func (d *Dispatcher) Handle(reqBody []byte) ([]byte, error) {
    x := bytes.TrimLeft(reqBody, " \t\r\n")
    if len(x) == 0 {
        return NewRPCResponse(nil, "2.0", nil, NewInvalidRequestError("Invalid json request")).Bytes()
    }

    if x[0] == '{' {
        var req Request
        if err := json.Unmarshal(reqBody, &req); err != nil {
            return NewRPCResponse(nil, "2.0", nil, NewInvalidRequestError("Invalid json request")).Bytes()
        }

        if req.Method == "" {
            return NewRPCResponse(req.ID, "2.0", nil, NewInvalidRequestError("Invalid json request")).Bytes()
        }

        resp, err := d.handleReq(req)

        return NewRPCResponse(req.ID, "2.0", resp, err).Bytes()
    }

    // handle batch requests
    var requests []Request
    if err := json.Unmarshal(reqBody, &requests); err != nil {
        return NewRPCResponse(nil, "2.0", nil, NewInvalidRequestError("Invalid json request")).Bytes()
    }

    responses := make([]Response, 0)
```

Figure 9 Source code of *Handle* function

1.3 RPC Sensitive Interface Permission

The vast majority of the RPC interfaces currently provided by dogechain are for querying data, and there are no high authority interfaces.

1.4 CLI commands security audit

The dogechain's CLI is mainly for nodes to perform related configurations, and its corresponding parameters are as follows:

```
Available Commands:
  backup      Create blockchain backup file by fetching blockchain data from the running node
  completion  Generate the autocompletion script for the specified shell
  genesis     Generates the genesis configuration file with the passed in parameters
  help        Help about any command
  ibft        Top level IBFT command for interacting with the IBFT consensus. Only accepts subcommands.
  license     Returns DogeChain-Lab Jury license and dependency attributions
  loadbot     Runs the loadbot to stress test the network
  monitor     Starts logging block add / remove events on the blockchain
  peers       Top level command for interacting with the network peers. Only accepts subcommands.
  secrets     Top level SecretsManager command for interacting with secrets functionality. Only accepts subcommands.
  server      The default command that starts the DogeChain-Lab Jury client, by bootstrapping all modules together
  status      Returns the status of the DogeChain-Lab Jury client
  txpool      Top level command for interacting with the transaction pool. Only accepts subcommands.
  version     Returns the current DogeChain-Lab Jury version
```

Figure 10 The dogechain node command line parameters

After testing, the relevant commands in the CLI meet the audit requirements and there is no security risk.

2 Node Security

2.1 Number of Node Connections

The dogechain nodes can use the server module to set parameters at startup, where max-inbound-peers, max-peers and max-peers limit the number of links to the node, and if the number of links exceeds the specified value, no links will be made.

```
Flags:
      --access-control-allow-origins stringArray   the CORS header indicating whether any JSON-RPC response can be shared with the specified origin (default [*])
      --block-gas-target string                    the target block gas limit for the chain. If omitted, the value of the parent block is used (default "0")
      --block-time uint                            minimum block time in seconds (default 2)
      --chain string                               the genesis file used for starting the chain (default "./genesis.json")
      --config string                              the path to the CLI config. Supports .json and .hcl
      --daemon                                     the flag indicating that the server ran as daemon
      --data-dir string                            the data directory used for storing DogeChain-Lab Jury client data (default "./jury-chain")
      --dns string                                 the host DNS address which can be used by a remote peer for connection
      --grpc-address string                        the GRPC interface (default "127.0.0.1:9632")
  -h, --help                                       help for server
      --jsonrpc string                             the JSON-RPC interface (default "0.0.0.0:8545")
      --libp2p string                              the address and port for the libp2p service (default "127.0.0.1:1478")
      --log-level string                           the log level for console output (default "INFO")
      --log-to string                              write all logs to the file at specified location instead of writing them to console
      --max-account-demotions uint                 maximum account demontion counter limit in the pool (default 10)
      --max-inbound-peers int                      the client's max number of inbound peers allowed (default 32)
      --max-outbound-peers int                     the client's max number of outbound peers allowed (default 8)
      --max-peers int                              the client's max number of peers allowed (default 40)
      --max-slots uint                             maximum slots in the pool (default 4096)
      --nat string                                 the external IP address without port, as can be seen by peers
      --no-discover                                prevent the client from discovering other peers (default: false)
      --price-limit uint                           the minimum gas price limit to enforce for acceptance into the pool (default 0)
      --prometheus string                          the address and port for the prometheus instrumentation service (address:port). If only port is defined (:port) it will bind to 0.0
.0.0:port
      --restore string                             the path to the archive blockchain data to restore on initialization
      --seal                                       the flag indicating that the client should seal blocks (default true)
      --secrets-config string                      the path to the SecretsManager config file. Used for Hashicorp Vault. If omitted, the local FS secrets manager is used
```

Figure 11 Node start-up parameters

After testing, the dogechain nodes can limit the number of connections to the current node using the server parameter (which defaults even if not specified), which can prevent the node from having too many connections.

```
]  jury.consensus.ibft.acceptState: Accept state: sequence=2500 round=1
]  jury.consensus.ibft: current snapshot: validators=4
]  jury.consensus.ibft: proposer calculated: proposer=0x8E27143EE92bA6fBE922d9628b0c4eC2A9AbdE0a block=2500
]  jury.network: Peer connected: id=16Uiu2HAkwrisKWxnuCLKqSXCQBQ3COvqeGUfqdRCfwwDGMQ5Wemp
]  jury.network: Peer disconnected: id=16Uiu2HAmFkPFbmm6PMtCrirNQT1GEza9ExvnGbjKneHTrX5cv2mm
]  jury.network: Attempted removing missing peer info 16Uiu2HAmFkPFbmm6PMtCrirNQT1GEza9ExvnGbjKneHTrX5cv2mm
]  jury.network: Peer connected: id=16Uiu2HAmFkPFbmm6PMtCrirNQT1GEza9ExvnGbjKneHTrX5cv2mm
```

Figure 12 Node connection information

2.2 Packet Size Limit

As shown in the figure below, the size of the data is checked when a transaction is received, which can avoid DoS attacks on nodes.

```
// constraints before entering the pool.
func (p *TxPool) validateTx(tx *types.Transaction) error {
    // Check the transaction size to overcome DOS Attacks
    if uint64(len(tx.MarshalRLP())) > txMaxSize {
        return ErrOversizedData
    }

    // Check if the transaction has a strictly positive value
    if tx.Value.Sign() < 0 {
        return ErrNegativeValue
    }
```

Figure 13 Source code for the *validateTX* function

2.3 Node Network Access Restrictions

Network parameters can be set when the node starts the chain, which can protect the node from attacks to some extent.

```
Flags:
      --access-control-allow-origins stringArray   the CORS header indicating whether any JSON-RPC response can be shared with the specified origin (default [*])
      --block-gas-target string                    the target block gas limit for the chain. If omitted, the value of the parent block is used (default "0")
      --block-time uint                            minimum block time in seconds (default 2)
      --chain string                               the genesis file used for starting the chain (default "./genesis.json")
      --config string                              the path to the CLI config. Supports .json and .hcl
      --daemon                                     the flag indicating that the server ran as daemon
      --data-dir string                            the data directory used for storing DogeChain-Lab Jury client data (default "./jury-chain")
      --dns string                                 the host DNS address which can be used by a remote peer for connection
      --grpc-address string                        the GRPC interface (default "127.0.0.1:9632")
  -h, --help                                       help for server
      --jsonrpc string                             the JSON-RPC interface (default "0.0.0.0:8545")
      --libp2p string                              the address and port for the libp2p service (default "127.0.0.1:1478")
      --log-level string                           the log level for console output (default "INFO")
      --log-to string                              write all logs to the file at specified location instead of writing them to console
      --max-account-demotions uint                 maximum account demontion counter limit in the pool (default 10)
      --max-inbound-peers int                      the client's max number of inbound peers allowed (default 32)
      --max-outbound-peers int                     the client's max number of outbound peers allowed (default 8)
      --max-peers int                              the client's max number of peers allowed (default 40)
      --max-slots uint                             maximum slots in the pool (default 4096)
      --nat string                                 the external IP address without port, as can be seen by peers
      --no-discover                                prevent the client from discovering other peers (default: false)
      --price-limit uint                           the minimum gas price limit to enforce for acceptance into the pool (default 0)
      --prometheus string                          the address and port for the prometheus instrumentation service (address:port). If only port is defined (:port) it will bind to 0.0.0.0:port
      --restore string                             the path to the archive blockchain data to restore on initialization
      --seal                                       the flag indicating that the client should seal blocks (default true)
      --secrets-config string                      the path to the SecretsManager config file. Used for Hashicorp Vault. If omitted, the local FS secrets manager is used
```

Figure 14 The server module parameters

3 Account &Asset Security

3.1 Account Model

The dogechain uses a similar account model to Ethereum, with the following basic data structure:

```
type Account struct {
    Nonce    uint64
    Balance  *big.Int
    Root     types.Hash
    CodeHash []byte
    Trie     accountTrie
}
```

Figure 15 The basic data structure of Account

- Nonce: The serial number of the transaction sent by the account;
- Balance: The balance of the account's platform coins;
- Root: The root hash of the storage state tree;
- CodeHash: The EVM code bound to the account;
- Trie: The root hash of the transaction lookup tree;

The dogechain does not implement the functions associated with the wallet module (e.g. creating accounts, signing transactions, backing up wallets).

3.2 Account Generation

(1) externally owned account

The dogechain does not provide a way to create external accounts directly, but the *secrets_init* interface is also used to create node accounts when creating nodes.

```go
func InitValidatorKey(secretsManager secrets.SecretsManager) (*ecdsa.PrivateKey, error) {
    // Generate the IBFT validator private key
    validatorKey, validatorKeyEncoded, keyErr := crypto.GenerateAndEncodePrivateKey()
    if keyErr != nil {
        return nil, keyErr
    }

    // Write the validator private key to the secrets manager storage
    if setErr := secretsManager.SetSecret(
        secrets.ValidatorKey,
        validatorKeyEncoded,
    ); setErr != nil {
        return nil, setErr
    }

    return validatorKey, nil
}
```

Figure 16 Source code of the *initValidatorKey* function

Tracing the relevant functions reveals that dogechain's account generation mechanism is the same as that of Ethereum, which will eventually generate the account private key using the secp256k1 elliptic curve algorithm and random numbers, and then calculate the corresponding public key and address.

```go
func GenerateKey() (*ecdsa.PrivateKey, error) {
    return ecdsa.GenerateKey(S256, rand.Reader)
}
```

Figure 17 Source code of the *GenerateKey* function (1)

```go
func GenerateKey(c elliptic.Curve, rand io.Reader) (*PrivateKey, error) {
    k, err := randFieldElement(c, rand)
    if err != nil {
        return nil, err
    }

    priv := new(PrivateKey)
    priv.PublicKey.Curve = c
    priv.D = k
    priv.PublicKey.X, priv.PublicKey.Y = c.ScalarBaseMult(k.Bytes())
    return priv, nil
}
```

Figure 18 Source code of the *GenerateKey* function (2)

Note that when using *secrets_init* module to generate an account, the private key of the account will be stored directly in plaintext in the specified path consensus/validator.key, although it is for generating node addresses and the node can be controlled for access, it is still recommended to store it encrypted.

(2) contract account

The figure below shows the two ways of creating contracts in dogechain. CREATE is a user created contract,

and the corresponding contract address is calculated based on the caller's address and the user's nonce; CREATE2 is generated by the caller, salt and contract bytecode hash. This is exactly the same as Ethereum.

```
if op == CREATE {
    address = crypto.CreateAddress(c.msg.Address, c.host.GetNonce(c.msg.Address))
} else {
    address = crypto.CreateAddress2(c.msg.Address, bigToHash(salt), input)
}
```

Figure 19 CREATE and CREATE2 directives

3.3 Transaction Signature

As shown in the figure below, dogechain implements functions related to transaction signature and verification signature, but due to the application scenario, it does not provide an external transaction signature interface, but it still verifies the signature when the transaction is verified.

```
func (e *EIP155Signer) SignTx(
    tx *types.Transaction,
    privateKey *ecdsa.PrivateKey,
) (*types.Transaction, error) {
    tx = tx.Copy()

    h := e.Hash(tx)

    sig, err := Sign(privateKey, h[:])
    if err != nil {
        return nil, err
    }

    tx.R = new(big.Int).SetBytes(sig[:32])
    tx.S = new(big.Int).SetBytes(sig[32:64])
    tx.V = new(big.Int).SetBytes(e.CalculateV(sig[64]))

    return tx, nil
}
```

Figure 20 Source code of the *SignTx* function

```go
func (e *EIP155Signer) Sender(tx *types.Transaction) (types.Address, error) {
    protected := true

    // Check if v value conforms to an earlier standard (before EIP155)
    bigV := big.NewInt(0)
    if tx.V != nil {
        bigV.SetBytes(tx.V.Bytes())
    }

    if vv := bigV.Uint64(); bits.Len(uint(vv)) <= 8 {
        protected = vv != 27 && vv != 28
    }

    if !protected {
        return (&FrontierSigner{}).Sender(tx)
    }

    // Reverse the V calculation to find the original V in the range [0, 1]
    // v = CHAIN_ID * 2 + 35 + {0, 1}
    mulOperand := big.NewInt(0).Mul(big.NewInt(int64(e.chainID)), big.NewInt(2))
    bigV.Sub(bigV, mulOperand)
    bigV.Sub(bigV, big35)

    sig, err := encodeSignature(tx.R, tx.S, byte(bigV.Int64()))
    if err != nil {
        return types.Address{}, err
    }
```

Figure 21 Source code of the *Sender* function

3.4 Asset Security

(1) Platform Asset Security

After a comprehensive audit, it is confirmed that there are only 2 ways to generate platform assets (DOGE tokens) in dogechain: premine, cross-chain bridge transfer, where premine is the address and initial quantity specified in the genesis file, and not repeatable premine (*ibft_switch* switches on consensus, but not again premine); and cross-chain transfer in, it will be minted to the target address because the system contract bridge will trigger special transactions (details in 6.2).

```go
func (e *Executor) WriteGenesis(alloc map[types.Address]*chain.GenesisAccount) types.Hash {
    snap := e.state.NewSnapshot()
    txn := NewTxn(e.state, snap)

    for addr, account := range alloc {
        if account.Balance != nil {
            txn.AddBalance(addr, account.Balance)
        }

        if account.Nonce != 0 {
            txn.SetNonce(addr, account.Nonce)
        }

        if len(account.Code) != 0 {
            txn.SetCode(addr, account.Code)
        }

        for key, value := range account.Storage {
            txn.SetState(addr, key, value)
        }
    }

    _, root := txn.Commit(false)

    return types.BytesToHash(root)
}
```

Figure 22 Source code of the *WriteGenesis* function

Likewise, there are two ways to reduce assets: cross-chain transfers and transaction fees.

The dogechain's fee mechanism is the same as Ethereum and was updated with the Istanbul upgrade during the audit.

(2) Contract Asset Security

The dogechain supports EVM in order to be compatible with Ether, and the contracts on it are not upgradable or modifiable.

4 Consensus Security

The dogechain uses the IBFT consensus mechanism and supports both PoA (Proof of Authority) and PoS (Proof-of-Stake) modes. In which in PoA, the validators are the ones responsible for creating the blocks and adding them to the block chain in a series.

All of the validators make up a dynamic validator-set, where validators can be added to or removed from the set by employing a voting mechanism. This means that validators can be voted in/out from the validators-set if the majority (51%) of the validator nodes vote to auth/drop a certain validator to/from the set. In this way, malicious validators can be recognized and removed from the network, while new trusted validators can be added to the network. All of the validators take turns in proposing the next block (round-robin), If the current

validator does not come out of the block properly, it will be moved down the list to the next validator.

In dogechain's PoS module, a new *epochSize* variable has been added that, at the end of each epoch, an epoch block is created, and after that event a new epoch starts. Validator sets are updated at the end of each epoch. Nodes query the validator set from the Staking Smart Contract during the creation of the epoch block, and save the obtained data to local storage. This query and save cycle is repeated at the end of each epoch (See 6.1 for details).

After feedback from the project, the IBFT-PoS consensus mechanism will be used when the mainnet goes online, so this report will only introduce the PoS consensus.

4.1 IBFT-PoS Consensus Process Analysis

IBFT-PoS consensus is divided into three main phases: PRE-PREPARE, PREPARE and COMMIT.

- PRE-PREPARE phase: The PRE-PREPARE phase is the first of three phases, indicating the start of a new round. In this phase, the Proposer node generates a proposed block and broadcasts it to all validator nodes. Then the Proposer node enters the PRE-PREPARED state. Other validator nodes receive a valid PRE-PREPARE message and enter the PRE-PREPARED state.
- PREPARE phase: In this phase, the validator node broadcasts PREPARE messages to other validator nodes and waits for +2/3 valid PREPARE messages to be received to enter the PREPARED state.
- COMMIT phase: In this phase, the validator node broadcasts a COMMIT message to other validator nodes and waits to receive +2/3 valid COMMIT messages to enter the COMMITTED state.

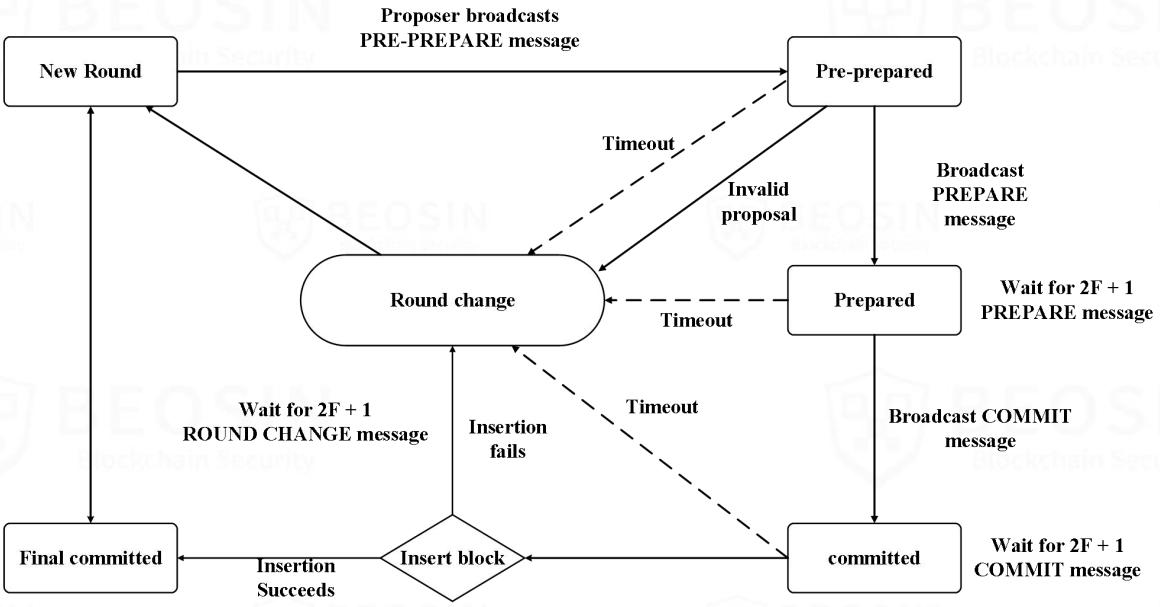


Figure 23 PoS consensus process

4.2 Validator Rotation Mechanism

In PoS mode, epoch mechanism is introduced, an epoch represents a consensus cycle, and its Validator is fixed within the same epoch. At present, the default size of dogechain epoch is 100,000 blocks, which means that every 100,000 blocks will enter a new cycle and reacquire the validator of the new cycle. At the code level, each time a new block is constructed, it will check whether the block height is the end block of the epoch, if yes, it means that the epoch has ended and will initiate a special transaction to get the current validators in the system contract validatorSet and act as validators for the new epoch.

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

Figure 24 Source code of the *IsLastOfEpoch* function

```
func QueryValidators(t TxQueryHandler, from types.Address) ([]types.Address, error) {
    method, ok := abis.ValidatorSetABI.Methods["validators"]
    if !ok {
        return nil, errors.New("validators method doesn't exist in Staking contract ABI")
    }

    selector := method.ID()
    res, err := t.Apply(&types.Transaction{
        From:     from,
        To:       &systemcontracts.AddrValidatorSetContract,
        Value:    big.NewInt(0),
        Input:    selector,
        GasPrice: big.NewInt(0),
        Gas:      queryGasLimit,
        Nonce:    t.GetNonce(from),
    })

    if err != nil {
        return nil, err
    }

    if res.Failed() {
        return nil, res.Err
    }

    return DecodeValidators(method, res.ReturnValue)
}
```

Figure 25 Source code of the *QueryValidators* function

Note:

(1) Any user holding DC tokens (system governance tokens) can pledge DC tokens to the system contract validatorSet to increase their pledge share.

(2) Only users whose pledged share is greater than the threshold specified in the validatorSet contract can be promoted by the contract owner to the validator for the next cycle.

(3) Users can be removed from the list of validators by the contract owner even if their pledged share is greater than the threshold specified in the validatorSet contract.

4.3 Rewards for Building Blocks

In dogechain, there is no block reward for the minter, only a transaction fee is sent to a specified coinbase address when the transaction is processed, which needs to be specified in the genesis file and is zero address by default.

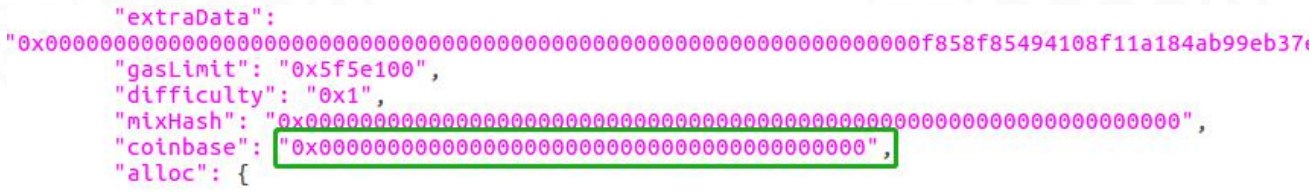


Figure 26 Partial information of the genesis file

4.4 Block Proposer Selection Algorithm Security Analysis

The following figure shows the core algorithm for calculating the proposer in the dogechain, which can be found to be based on the index of the address in the validator list for sequential blocking, and if the corresponding validator is not properly blocked, it will be deferred to the next one.

```
func (v *ValidatorSet) CalcProposer(round uint64, lastProposer types.Address) types.Address {
    var seed uint64

    if lastProposer == types.ZeroAddress {
        seed = round
    } else {
        offset := 0
        if indx := v.Index(lastProposer); indx != -1 {
            offset = indx
        }

        seed = uint64(offset) + round + 1
    }

    pick := seed % uint64(v.Len())

    return (*v)[pick]
}
```

Figure 27 Source code of the f *CalcProposer* function

5 Transaction Model Security

5.1 Transaction Processing Flow

The dogechain's transaction processing is divided into 2 main parts: adding transactions to the pool and executing transactions. Adding a transaction to the pool means that each node adds the submitted transaction to the pool. In addition to the usual checks, adding a transaction in dogechain also checks whether the transaction reaches the minimum price *priceLimit* of the node.

```
// If the from field is set, check that
// it matches the signer
if tx.From != types.ZeroAddress &&
    tx.From != from {
    return ErrInvalidSender
}

// If no address was set, update it
if tx.From == types.ZeroAddress {
    tx.From = from
}

// Reject underpriced transactions
if tx.IsUnderpriced(p.priceLimit) {
    return ErrUnderpriced
}

// Grab the state root for the latest block
stateRoot := p.store.Header().StateRoot

// Check nonce ordering
if p.store.GetNonce(stateRoot, tx.From) > tx.Nonce {
    return ErrNonceTooLow
}
```

Figure 28 Partial source code of transaction check

Transaction execution, is when the block is constructed, the miner will execute the transactions within the block. Also, if the corresponding transaction triggers a Deposited or Withdrawn on the system contract bridge (i.e. indicating a cross-chain operation), it will be handled specially, and the source code is as follows.

```
for _, log := range logs {
    if len(log.Topics) == 0 {
        continue
    }

    switch log.Topics[0] {
    case bridge.BridgeDepositedEventID:
        parsedLog, err := bridge.ParseBridgeDepositedLog(log)
        if err != nil {
            return err
        }

        t.state.AddBalance(parsedLog.Receiver, parsedLog.Amount)
    case bridge.BridgeWithdrawnEventID:
        parsedLog, err := bridge.ParseBridgeWithdrawnLog(log)
        if err != nil {
            return err
        }

        // the total one is the real amount of Withdrawn event
        realAmount := big.NewInt(0).Add(parsedLog.Amount, parsedLog.Fee)

        if err := t.state.SubBalance(parsedLog.Contract, realAmount); err != nil {
            return err
        }

        // the fee goes to system Vault contract
        t.state.AddBalance(systemcontracts.AddrVaultContract, parsedLog.Fee)
    }
}
```

Figure 29 Special transaction processing for cross-chain bridges

5.2 Replay Attack

The replay attack is when a block chain undergoes a hard fork, the block chain permanently diverges and creates two chains with exact correspondence of historical transactions, addresses, private keys and balances, and the same transaction can be validated on both chains at the same time. Because dogechain has similar modules in order to be compatible with Ethereum, the focus is on whether to replay transactions with Ether and similar chains.

After testing, in the 0.4.0 version of dogechain, its transaction signature does not contain the chainId, then other transactions of similar Ethereum chains can complete the replay attack on dogechain, but it is fixed in the 0.4.1 version of the commit.

```go
// calcTxHash calculates the transaction hash (keccak256 hash of the RLP value)
func calcTxHash(tx *types.Transaction, chainID uint64) types.Hash {
    a := signerPool.Get()

    v := a.NewArray()
    v.Set(a.NewUint(tx.Nonce))
    v.Set(a.NewBigInt(tx.GasPrice))
    v.Set(a.NewUint(tx.Gas))

    if tx.To == nil {
        v.Set(a.NewNull())
    } else {
        v.Set(a.NewCopyBytes((*tx.To).Bytes()))
    }

    v.Set(a.NewBigInt(tx.Value))
    v.Set(a.NewCopyBytes(tx.Input))

    // EIP155
    if chainID != 0 {
        v.Set(a.NewUint(chainID))
        v.Set(a.NewUint(0))
        v.Set(a.NewUint(0))
    }

    hash := keccak.Keccak256Rlp(nil, v)

    signerPool.Put(a)

    return types.BytesToHash(hash)
}
```

Figure 30 Source code of the *calcTxHash* function

5.3 Dusting Attack

The dusting attack is when an attacker sends a very small amount of tokens, called "dust", to a user's wallet. By tracking the dusted wallet funds and all transactions, the attacker can then connect to these addresses and determine the company or person to whom these wallet addresses belong, destroying the anonymity of the block chain; or misuse the block chain resources, causing the block chain memory pool strain. If the dust money is not moved, the attacker cannot establish a connection to it and cannot complete the de-anonymization of the wallet or address owners.

The dogechain uses an account model, unlike Bitcoin's UTXO model, where the "balance" of a user's wallet consists of a number of unspent transaction outputs, and the dust UTXO is always represented in the user's transfer transactions when the user transfers money, Bitcoin's transactions consist of inputs Bitcoin transactions are composed of inputs and outputs, so transactions can be strung together through UTXO to achieve de-anonymization through dust attacks. In dogechain, after the dust money is sent to the user's wallet, the amount is added to the user's balance, which is not independent of the user's balance, and the attacker cannot achieve the purpose of de-anonymization. Moreover, each transaction in dogechain consumes a certain amount of gas, and the corresponding nodes can also set the min.

5.4 Trading Flooding Attacks

In dogechain, transactions are subject to a fee. The base fee for creating a contract is 53,000, and the base fee for a normal transaction is 21,000.

```
const (
    spuriousDragonMaxCodeSize = 24576

    TxGas                 uint64 = 21000 // Per transaction not creating a contract
    TxGasContractCreation uint64 = 53000 // Per transaction that creates a contract
)
```

Figure 31 Transactions minimum gas consumption

However, it is worth mentioning that in the current test environment, if a node does not specify a minimum gas price limit, then the default will be 0, indicating that the transaction will not consume gas. Therefore, it is recommended that each node specify a price limit when dogechain goes live.

5.5 Double-spending Attack

For dogechain's transactions, each transaction of the account contains a unique and mintable nonce, and dogechain uses the consensus model of IBFT-PoS, it is also difficult to complete a double-spending through a 51% attack.

```
xing
    raise ValueError(response["error"])
ValueError: {'code': -32600, 'message': 'already known'}
```

Figure 32 Repeat transaction error message

```
    raise ValueError(response["error"])
ValueError: {'code': -32600, 'message': 'nonce too low'}
```

Figure 33 Invalid transaction error message

5.6 Illegal Transactions

The user will sign the entire transaction data when initiating a transaction, and any changes to the data in the transaction will cause the dogechain node to fail the signature check.

```
// Check if v value conforms to an earlier standard (before EIP155)
bigV := big.NewInt(0)
if tx.V != nil {
    bigV.SetBytes(tx.V.Bytes())
}

if vv := bigV.Uint64(); bits.Len(uint(vv)) <= 8 {
    protected = vv != 27 && vv != 28
}

if !protected {
    return (&FrontierSigner{}).Sender(tx)
}

// Reverse the V calculation to find the original V in the range [0, 1]
// v = CHAIN_ID * 2 + 35 + {0, 1}
mulOperand := big.NewInt(0).Mul(big.NewInt(int64(e.chainID)), big.NewInt(2))
bigV.Sub(bigV, mulOperand)
bigV.Sub(bigV, big35)

sig, err := encodeSignature(tx.R, tx.S, byte(bigV.Int64()))
if err != nil {
    return types.Address{}, err
}

pub, err := Ecrecover(e.Hash(tx).Bytes(), sig)
if err != nil {
    return types.Address{}, err
}

buf := Keccak256(pub[1:])[12:]

return types.BytesToAddress(buf), nil
```

Figure 34 Transaction signature check

If the dogechain node is malicious, the signature checks fails here, but the verification node also checks the transaction signature again, and the forged transaction will fail to be verified.

```
jury.consensus.ibft: failed to bulk sync: err="failed to write bulk sync blocks: invalid merkle root"
jury.blockchain: write block: num=1136 parent=0xa52f159cdee7d6e5bbfc27bf27b967bf3727438a52360134d908a0c08965dccc
jury.consensus.ibft: failed to bulk sync: err="failed to write bulk sync blocks: invalid merkle root"
jury.blockchain: write block: num=1136 parent=0xa52f159cdee7d6e5bbfc27bf27b967bf3727438a52360134d908a0c08965dccc
jury.consensus.ibft: failed to bulk sync: err="failed to write bulk sync blocks: invalid merkle root"
jury.blockchain: write block: num=1136 parent=0xa52f159cdee7d6e5bbfc27bf27b967bf3727438a52360134d908a0c08965dccc
jury.consensus.ibft: failed to bulk sync: err="failed to write bulk sync blocks: invalid merkle root"
jury.blockchain: write block: num=1136 parent=0xa52f159cdee7d6e5bbfc27bf27b967bf3727438a52360134d908a0c08965dccc
jury.consensus.ibft: failed to bulk sync: err="failed to write bulk sync blocks: invalid merkle root"
jury.blockchain: write block: num=1136 parent=0xa52f159cdee7d6e5bbfc27bf27b967bf3727438a52360134d908a0c08965dccc
jury.consensus.ibft: failed to bulk sync: err="failed to write bulk sync blocks: invalid merkle root"
jury.blockchain: write block: num=1136 parent=0xa52f159cdee7d6e5bbfc27bf27b967bf3727438a52360134d908a0c08965dccc
jury.consensus.ibft: failed to bulk sync: err="failed to write bulk sync blocks: invalid merkle root"
jury.blockchain: write block: num=1136 parent=0xa52f159cdee7d6e5bbfc27bf27b967bf3727438a52360134d908a0c08965dccc
jury.consensus.ibft: failed to bulk sync: err="failed to write bulk sync blocks: invalid merkle root"
jury.blockchain: write block: num=1136 parent=0xa52f159cdee7d6e5bbfc27bf27b967bf3727438a52360134d908a0c08965dccc
jury.consensus.ibft: failed to bulk sync: err="failed to write bulk sync blocks: invalid merkle root"
```

Figure 35 Abnormal node block error message

5.7 Fake Deposit Attack

Fake recharge attack is the transaction execution failure, but the corresponding receipt status shows success. In dogechain, there are only two types of transaction status, failed(0x0) and Success(0x1). For transactions that pass the check, both gas exceeds the block limit and transaction execution fails will return failed, and there is no chain-level fake deposit attack.


```
{
    "jsonrpc": "2.0",
    "id": 1,
    "result": {
        "root": "0x0000000000000000000000000000000000000000000000000000000000000000",
        "cumulativeGasUsed": "0x5726",
        "logsBloom":
            "0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
            0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
            0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
        "logs": [],
        "status": "0x1",
        "transactionHash": "0x51e96a59c0bbf1db9701ffcf7c626057422bd333ec1e53dbdcb2a2b9d9291171",
        "transactionIndex": "0x0",
        "blockHash": "0x0ae81a539309dcd4e108f3bf689df81e31a05e69c7f8168b4a104219b0cfe2a1",
        "blockNumber": "0x2c1f",
        "gasUsed": "0x5726",
        "contractAddress": null,
        "from": "0x04DAbf4e9f28B7cdDFfDd8af49a33aC6C901403c",
        "to": "0x0000000000000000000000000000000000001001"
    }
```


Figure 36 Transaction receipt information

5.8 Transaction Pool Security Audit

This item mainly audits the sequencing of the transaction pool to avoid too many future transactions blocking the pool.

After testing, there are some problems with dogechain's transaction pool processing, please see 2.7.3 for details

6 System Contract Security

The dogechain system has three contracts: validatorSet, bridge and vault. These contracts will be deployed when the chain is launched, and the addresses are： 0x0000000000000000000000000000000000001001, 0x0000000000000000000000000000000000001002 and 0x0000000000000000000000000000000000001003. These three system contracts are created when the first block is packed. (The link to the contract is: https://github.com/dogechain-lab/jury-contracts. An audit has been completed by our team's contract audit team and the link to the audit is: https:// beosin.com/audits/jury_202204281131.pdf).

```go
// Predeploy ValidatorSet smart contract if needed
if p.shouldPredeployValidatorSetSC() {
    account, err := p.predeployValidatorSetSC()
    if err != nil {
        return err
    }

    chainConfig.Genesis.Alloc[systemcontracts.AddrValidatorSetContract] = account
}

// Predeploy bridge contract
if bridgeAccount, err := p.predeployBridgeSC(); err != nil {
    return err
} else {
    chainConfig.Genesis.Alloc[systemcontracts.AddrBridgeContract] = bridgeAccount
}

// Predeploy vault contract if needed
if p.shouldPredeployValidatorSetSC() {
    vaultAccount, err := p.predeployVaultSC()
    if err != nil {
        return err
    }

    chainConfig.Genesis.Alloc[systemcontracts.AddrVaultContract] = vaultAccount
}
```

Figure 37 System contract pre-deployment

6.1 validatorSet Contract

(1) Basic functions of the validatorSet contract

The validatorSet contract can be pledged by users holding governance tokens (DCs), and then the contract owner designates the addresses that reach the pledged amount to become validators, and when the next epoch opens, the dogechain will acquire these validators as block-out nodes for the new epoch.

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

function deleteValidator(address account) external onlyOwner {
  require(_validators.length > _minimum, "Validators can't be less than minimum");
  require(_addressToIsValidator[account], "Account must be validator");
  _deleteFromValidators(account);
  emit ValidatorDeleted(msg.sender, account);
}
```

Figure 38 Part source code of *validatorSet* contract

(2) validatorSet contract security testing

This test focuses on the impact of the contract on dogechain, not the contract itself.

◆ Whether the function of adding removing is effective

After testing, the add/remove validator function in the validatorSet contract meets the design requirements, and the next epoch takes effect. However, it should be noted that even if the pledge amount of a node reaches the specified threshold, the owner of the validatorSet contract needs to call the *addValidator* function to add it to become a validator; and the owner can remove the node whose pledge amount reaches the threshold directly.

In addition, the value of the _threshold variable in the validatorSet contract in the test version is 0. It is recommended to upgrade it as soon as possible.

◆ Repeatedly add the same address

If the pledge amount of the node address reaches the threshold, it is possible to be repeatedly added to the validator list by the owner. It is recommended to check whether the address is repeated when adding the validator.

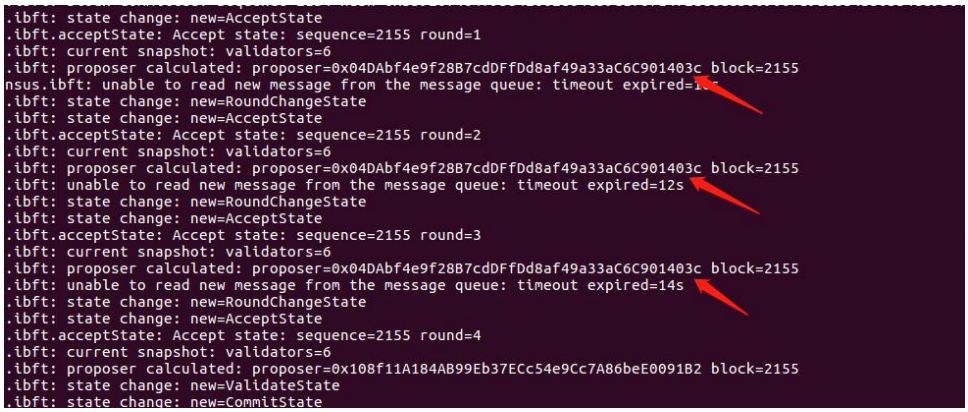


Figure 39 Repeat validator's building block information

(3) If the add address is an invalid node address, then the outgoing block of the corresponding node will be carried over to the next verifier node due to invalid expiration.

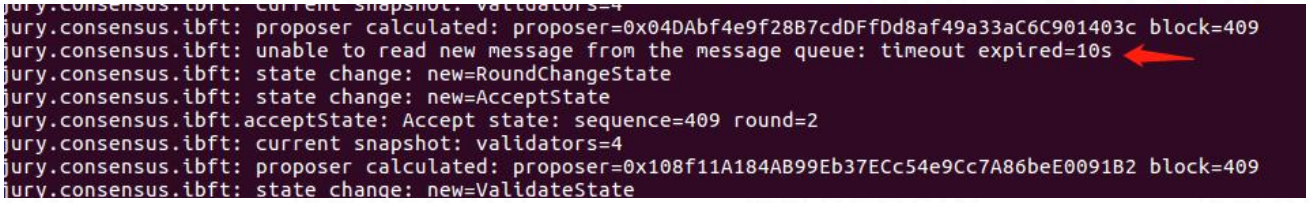


Figure 40 Invalid validator's building block information

6.2 bridge Contract

(1) Basic functions of the bridge contract

The figure below shows the cross-chain core function of the bridge contract. When the specified token is burned in the cross-chain bridge contract on another EVM chain by the user, the signer of this contract will call the *deposit* function of this contract for cross-chain signing based on the corresponding transaction. Once the number of signer signatures reaches the threshold, the Deposited event will be triggered and dogechain will capture the event and send a DOGE to the target address in that event.

```
function deposit(address receiver, uint256 amount, string memory txid, string memory sender) external onlySigner {
  bytes32 key = keccak256(abi.encodePacked(receiver, amount, txid, sender));
  Order storage order = orders[key];
  if (order.finished) {
    return;
  }

  for (uint256 i = 0; i < order.signers.length; i++) {
    if (order.signers[i] == msg.sender) {
      return;
    }
  }

  order.receiver = receiver;
  order.amount = amount;
  order.sender = sender;
  order.txid = txid;
  order.signers.push(msg.sender);

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

Figure 41 Part source code of bridge contract

```
for _, log := range logs {
    if len(log.Topics) == 0 {
        continue
    }

    switch log.Topics[0] {
    case bridge.BridgeDepositedEventID:
        parsedLog, err := bridge.ParseBridgeDepositedLog(log)
        if err != nil {
            return err
        }

        t.state.AddBalance(parsedLog.Receiver, parsedLog.Amount)
```

Figure 42 Handling of Deposited event

When users need to transfer assets from dogechain to other chains, they need to call the *withdraw* function of bridge system contract, send the DOGE that needs to cross chain to bridge system contract, dogechain will also get the Withdrawn event, burn this part of DOGE, and send the fee collected to vault system contract.

```
function withdraw(string memory receiver) external payable {
  require(msg.value >= _MinimumThreshold, "Forbid");
  uint256 fee = msg.value.mul(_rate).div(1e4);
  uint256 amount = msg.value.sub(fee);
  _totalSupply = _totalSupply.sub(amount);
  emit Withdrawn(msg.sender, amount, fee, receiver);
}
```

Figure 43 Part source code of bridge contract

```
case bridge.BridgeWithdrawnEventID:
    parsedLog, err := bridge.ParseBridgeWithdrawnLog(log)
    if err != nil {
        return err
    }

    // the total one is the real amount of Withdrawn event
    realAmount := big.NewInt(0).Add(parsedLog.Amount, parsedLog.Fee)

    if err := t.state.SubBalance(parsedLog.Contract, realAmount); err != nil {
        return err
    }

    // the fee goes to system Vault contract
    t.state.AddBalance(systemcontracts.AddrVaultContract, parsedLog.Fee)
}
```

Figure 44 Handling of Withdrawn event

(2) bridge contract security testing

- Fake trigger events

Create the contract and trigger the Deposited event after calling the bridge contract. After testing, it is not possible to trigger minting tokens by faking the Deposited event.

```
}
event Deposited(address indexed receiver, uint256 indexed amount, string txid, string sender);


function evnetTest() public payable {
    uint valueTemp = msg.value;
    string memory addr = toAsciiString(msg.sender);
    a(bridgeC).withdraw{value:valueTemp}(addr);
    emit Deposited(msg.sender, valueTemp, txid, addr);
}
```

Figure 45 Tested Attack Contract

- Multiple withdraws for one transaction

Tested and not repeatable withdraw assets.

6.3 vault Contract

(1) Basic functions of the vault contract

The vault contracts are contracts used by the dogechain to store cross-chain fees, which can be withdrawn by the owner of the Vault contract.

```
case bridge.BridgeWithdrawnEventID:
    parsedLog, err := bridge.ParseBridgeWithdrawnLog(log)
    if err != nil {
        return err
    }

    // the total one is the real amount of Withdrawn event
    realAmount := big.NewInt(0).Add(parsedLog.Amount, parsedLog.Fee)

    if err := t.state.SubBalance(parsedLog.Contract, realAmount); err != nil {
        return err
    }

    // the fee goes to system Vault contract
    t.state.AddBalance(systemcontracts.AddrVaultContract, parsedLog.Fee)
}
```

Figure 46 Related logic of vault contract

Disclaimer

The Audit Report issued by Beosin is related to the services agreed in the relevant service agreement. The Project Party or the Served Party (hereinafter referred to as the "Served Party") can only be used within the conditions and scope agreed in the service agreement. Other third parties shall not transmit, disclose, quote, rely on or tamper with the Audit Report issued for any purpose.

The Audit Report issued by Beosin is made solely for the code, and any description, expression or wording contained therein shall not be interpreted as affirmation or confirmation of the project, nor shall any warranty or guarantee be given as to the absolute flawlessness of the code analyzed, the code team, the business model or legal compliance.

The Audit Report issued by Beosin is only based on the code provided by the Served Party and the technology currently available to Beosin. However, due to the technical limitations of any organization, and in the event that the code provided by the Served Party is missing information, tampered with, deleted, hidden or subsequently altered, the audit report may still fail to fully enumerate all the risks.

The Audit Report issued by Beosin in no way provides investment advice on any project, nor should it be utilized as investment suggestions of any type. This report represents an extensive evaluation process designed to help our customers improve code quality while mitigating the high risks in Block chain.

BEOSIN
Blockchain Security

Official Website

https://www.beosin.com

Telegram

https://t.me/+dD8Bnqd133RmNWNl

Twitter

https://twitter.com/Beosin_com

Email

Contact@beosin.com