Crypto.com Chain

Welcome to Crypto.com Chain's documentation!

Get Started →

The high-level design vision is documented in

1. whitepaper and
2. technical whitepaper.

For a bit clearer technical vision, see design philosophy.

The documentation in this repository site is meant to provide specifications and implementation details that will be useful to third party developers or contributors to the main repository.

Getting Started

This is the beginner's tutorial to boot-strap Chain

Pre-requisites

Prepare ubuntu 18.0.x and Intel CPU

```
~/bin
```

bin folder is the main folder for binaries

How it works

chain-tx-enclave (use intel sgx) -> working as dockerchain-abci (connects to enclave) tendermint (connects to chain-abci)

Compile enclave

1. git clone chain-tx-enclave, activate docker

```
cd ~
git clone https://github.com/crypto-com/chain-tx-enclave.git
cd chain-tx-enclave
docker build -t chain-tx .
docker run -ti --rm -p 25933:25933 -v ~/chain-tx-enclave/:/root/sgx -it chain-tx /bin/
```

2. Inside docker, install necessary components

```
apt update
apt install rsync curl git gcc unzip  libzmq3-dev
apt install libsnappy-dev wget vim pkg-config
```

3. Install sdk. You can install intel-sgx-sdk, and compile the application.

```
cd sgx
export SGX_MODE=SW
export NETWORK_ID=ab
make
```

4. Build sgx application

```
cd ~/sgx/app
cargo build
```

For hw

```
docker run -ti --device /dev/isgx -v ~/chain-tx-enclave/:/root/sgx -it chain-tx /bin/b

root@docker:/# LD_LIBRARY_PATH=/opt/intel/libsgx-enclave-common/aesm /opt/intel/libsgx
```

5. Copy enclave lib file to the system, and cargo build if build fails, please copy libEnclave_u.a manually.

```
cd ~/sgx
make
cd ~/sgx/app
cp libEnclave_u.a /usr/local/lib
```

6. Compile the release binary

```
cargo build --release
cp ./target/release/tx-validation-app  ~/bin
```

Congratulations! chain-tx-enclave is now ready to go!

Setup path

cd ~ vi .profilemkdir ~/bin

```
export PATH="$HOME/.cargo/bin:$PATH"
export PATH=$HOME/bin:$PATH
```

Install Rust

```
cd ~curl https://sh.rustup.rs -sSf | shsource $HOME/.cargo/envrustup default nightly-2
```

Install Tendermint

```
cd ~/binwget https://github.com/tendermint/tendermint/releases/download/v0.32.1/tender
```

Get source code

```
git clone https://github.com/crypto-com/chain.git
```

Compile chain-abci

```
vi ~/.cargo/config
```

```
[build]
rustflags = ["-Ctarget-feature=+aes,+ssse3"]
```

```
:wqcargo build -release cp ./target/release/chain-abci ~/bin
```

Run the program

enclave

```
docker run -ti -rm -p 25933:25933 -v ~/chain-tx-enclave:/root/sgx -it linux /bin/bashe
```

abci

tendermint

```
tendermint init tendermint genesis.json
```

```
{
"genesis_time": "2019-03-21T02:26:51.366017Z",
"chain_id": "test-chain-y3m1e6-AB",
  "consensus_params": {
    "block": {
      "max_bytes": "22020096",
      "max_gas": "-1",
      "time_iota_ms": "1000"
    },
    "evidence": {
      "max_age": "100000"
    },
    "validator": {
      "pub_key_types": [
        "ed25519"
      ]
    }
  },
  "validators": [
    {
      "address": "2962FC8AAE279DB2D2AB583926A89D8738EB8B28",
      "pub_key": {
        "type": "tendermint/PubKeyEd25519",
        "value": "DHGjayKp5xt0BBvZI0EMC6fr+OLgAsHFs+97bz7lShg="
      },
      "power": "10",
      "name": ""
    }
  ],

"app_hash": "FAF9E47D07382ADEC643BA24561F8E1C6D61FE575D184265C443B2637355CA25",
"app_state": {"distribution":{"0x0db221c4f57d5d38b968139c06e9132aaf84e8df":["250000000
}
```

```
tendermint node
```

client rpc

Check working

```
curl 'http://localhost:26657/abci_query?path=0x6163636f756e74&data=0x0db221c4f57d5d38b
```

Other information

github.com/crypto-com/chain/blob/master/README.md

Consensus →

Consensus

Crypto.com Chain prototype uses Tendermint Core as its consensus algorithm.

It utilizes the ABCI (Application BlockChain Interface).

This allows "plugging" custom applications with Tendermint.

If the application is written in Go, it can be linked directly; if it's written in other languages, it communicates over 3 TCP or Unix sockets.

The details of this interface can be found on: tendermint.com/docs/spec/abci/abci.html

As Crypto.com Chain Core code is written in Rust, we utilize (and aim to continually improve) the rust-abci library.

What is executed when by the consensus engine and by the ABCI application can be seen in this diagram.

Client: Interacting with the blockchain

To query a blockchain or submit a transaction, one can use the Tendermint RPC for that.

The details of the RPC mechanism can be found on: tendermint.com/rpc/

Currently, it supports 3 methods:

1. URI over HTTP
2. JSON-RPC over HTTP
3. JSON-RPC over WebSockets

The RPC HTTP server is executed on every full node. The RPC methods are equivalent, but WebSockets allow realtime subscription to different events.

Note that Tendermint RPC is for internal use only, as it doesn’t support rate-limiting, authentication etc., so it shouldn’t be directly exposed to the internet.

At this moment, it’s also recommended to use `tendermint lite` as a local proxy for the Chain client libraries when connecting to remote full nodes.

The Chain client interaction with Tendermint is currently done via a custom `client-rpc` crate.

broadcast_tx_(a)sync

This method takes "tx" parameter which is application-specific binary data (see transaction serialization for details on Chain binary format). The transaction binary payload is either hex-encoded (when called with the URI method) or base64-encoded (when called with JSON-RPC).

abci_query

Currently the main usage is that given a path "account", one can query the current "staked state" of some address (which is provided as the "data" field).

APP HASH

Tendermint expects the ABCI application to be deterministic and consistency is checked that each instance, given the same input (block/consensus events + transaction data), updates its state in the same way and calculates the same "application hash" which is a compact representation of the overall ABCI application state.

In Chain, it is a Blake2s hash of several components:

- root of a Merkle tree of a valid transactions in a given block
- root of a sparse Merkle trie of staked states (see accounting details)
- binary serialized state of rewards pool (see serialization for details on Chain binary format and genesis for details on "state")

Conventions

As genesis information is taken from the Ethereum network, the same address format is used (i.e. hexadecimal encoding of 20-bytes from a keccak-256 hash of a secp256k1 public key).

For Tendermint data, its conventions must be followed (e.g. validator keys are Ed25519, base64-encoded … "addresses" are the first 20 bytes of SHA256 of the raw public key bytes).

For Crypto.com Chain, it has the following conventions:

- Chain-ID: this is a string in Tendermint's genesis.json. In Crypto.com Chain, it should end with two hex digits.
- Network-ID: a single byte determined by the two last hex digits of Chain-ID. It is included in metadata of every transaction
- Transactions, addresses etc.: see transaction binary serialization, accounting model, addresses / witness and format / types

← Getting Started

Genesis →

Genesis

The initial state of the network is started from the existing ERC20 contract on Ethereum. On the Ethereum mainnet, it is 0xA0b73E1Ff0B80914AB6fe0444E65848C4C34450b.

The current genesis procedure is the following:

1. The initial set of validator (council node) identities is established: each council node should have an associated validator public key (from the Tendermint Ed25519 keypair, used for signing blocks) and initial staking address (same as Ethereum), …
2. The initial network parameters (e.g. minimum stake amounts) are chosen
3. At a determined time, the list of ERC20 holders is snapshotted / computed from the Ethereum network.
4. The list should contain three dedicated addresses for:

- Secondary distribution and launch incentive (0x20a0bee429d6907e556205ef9d48ab6fe6a55531 and 0x35f517cab9a37bc31091c2f155d965af84e0bc85 on the Ethereum mainnet)
- Long term incentive (0x71507ee19cbc0c87ff2b5e05d161efe2aac4ee07 on the Ethereum mainnet)

The balances of these dedicated addresses are put to the initial “Rewards Pool” (where transaction fees are paid to and network operation rewards are paid from).

1. For every ERC20 holder address in the initial distribution, besides the above three addresses, the following rules are used for computing the initial genesis state:

- If the address is owned / controlled by a smart contract, its balance goes to the initial “Rewards Pool”

> WARNING
>
> DEX, multisig etc. contracts should be avoided during the Step 3

- If the address is an externally owned account and corresponds to the initial staking address of one of council nodes, its balance starts as “bonded” in the corresponding staked state (see accounting mode for more details).
- Otherwise (i.e. the address is an externally owned account, but not of any validators), the address balance starts as “unbonded” in the corresponding staked state (see accounting mode for more details).

need to be signed by the developer production keys).

Tendermint extra information

As described in consensus, Tendermint executes with the application-specific code via ABCI.

The genesis information (network parameters, initial validators etc.) is set in the `app_data` field in genesis.json – this information is then passed to the ABCI application in the InitChainRequest.

← Consensus

Transaction Accounting Model →

Transaction Accounting Model

Crypto.com Chain uses both Unspent Transaction Outputs (UTXOs) and (a form of) accounts for its accounting model, inspired by the work on chimeric ledgers.

Motivation

The native token used in Crypto.com Chain serves two main purposes:

1. Payments
2. Network operation (staking etc.)

These two realms have different rules and properties. These differences are highlighted in the table below:

	Transaction Volume	Visibility	State Changes	Value Transfer Rules
Payments	High	Minimal / confidentiality / data protection is desired	UTXO set is the only “state”: changes only by transactions	Flexible: new address per invoice, ad-hoc n-of-m address formations (e.g. for escrows); encoding extra information for atomic swaps etc.
Network operation (staking etc.)	Low	Maximal transparency is desired	Both by transactions and network events (e.g. a validator not following the protocol)	Self-contained: same account changes (reward payouts, unbonding…)

UTXO+Accounts model

For this reason, we chose the mixed model where:

- UTXO is used for payments / value transfers
- Account-model is used for network operations (“staked state”)

Different types of transactions and how they relate to these accounting are described in transactions.

Staked state

The current account usage is self-contained limited. Each account (“staked state”) contains two balances:

- bonded amount
- unbonded amount

The bonded amount is the amount used to check against minimal staking requirements and used to calculate the Tendermint validator voting power (in case of council nodes).

As it may take time for the network evidence of malicious activity (e.g. double signing) to appear, the stake cannot be withdrawn immediately and is first moved to the “unbonded” balance.

The unbonded balance can be withdrawn (into transaction outputs) after “unbonded_from” time if the account was not jailed / slashed (see staking).

Staked State Storage

The account state is currently stored in a sparse Merkle trie structure (currently MerkleBIT backed by RocksDB, but it may be migrated to a more robust / better understood structure, e.g. a Patricia Merkle trie or IAVL+).

← Genesis

Transaction →

Transaction

Transaction Identifier

Each transaction has an identifier (typically shortened as TX ID). It is currently defined as

```
blake2s_hash(SCALE-encoded transaction binary data)
```

See serialization for more details about the transaction binary format.

Note: the initial prototype uses blake2s, but it may be later changed to blake2b or something more complex: e.g. transaction identifier is a root of a Merkle tree formed from different transaction components as leaves

Witness

See signature-schemes for more details

Textual Address Representation

Crypto.com Chain supports threshold / multi-signature addresses that are represented as a single hash (see signature-schemes) which is different from Ethereum.

To represent the underlying byte array in a textual form, Bech32 is used. The convention for the human-readable part is the following:

- cro: mainnet payment
- tcro: testnet payment
- dcro: local devnet/regtest payment
- staking addresses (see accounting) are textually represented in hexadecimal encoding to match the initial Ethereum ones

Transaction Fees

The initial prototype uses a linear fee system, see staking for details.

Transaction Types

Basic Types (unencrypted / non-shielded):

> **NOTE**
>
> All these types should also contain metadata, such as network ID

- TransferTx: UTXO-based transfer of funds, takes UTXO inputs, creates UTXO outputs - fee
- DepositStakeTx: takes UTXOs inputs, deposits them in the specified account's bonded amount - fee
- UnbondStakeTx: takes nonce, amount, account and moves in the same account tokens from bonded to unbonded with timelock - fee
- WithdrawUnbondedTx: takes nonce, account and creates UTXOs - fee

Advanced Types:

- AddCouncilNodeTx: takes council node data, staking address; co-signed by 2/3 current nodes
- EditCouncilNodeTx: takes council node data, signed by that node
- RemoveCouncilNodeTx: takes council node id; co-signed by 2/3 current nodes
- ChangeNetworkParamTX?
- UnjailTx: takes nonce, account, signed by the account's corresponding key
- AddWhitelistServiceNodeTx: takes node data, whitelist type (customer acquirer, merchant acquirer, settlement agent), staking address; co-signed by 2/3 current nodes
- EditWhitelistServiceNodeTx: takes node data, signed by that node
- RemoveWhitelistServiceNodeTx: takes whitelisted node id; co-signed by 2/3 current nodes
- AddMerchantIdTx: takes merchant data (certificate + cert-signed pk or some payment gateway point?), signed by merchant acquirer
- RemoveMerchantIdTx: takes merchant id, signed by merchant acquirer

← Transaction Accounting Model

Serialization →

Serialization

After several iterations of binary formats, Crypto.com Chain settled on using the SCALE (Simple Concatenated Aggregate Little-Endian) codec.

It is formally described in Section B.1 of Polkadot RE Protocol Specification.

← Transaction

Signature Schemes →

Signature Schemes

There are two address types which require corresponding signature schemes:

1. "RedeemAddress": Ethereum account address (for ERC20 reedem / backwards-compatibility); see `init/address.rs` in `chain-core` .
2. "Tree": threshold multisig addresses; records a root of a "Merklized Abstract Syntax Tree" where branches are "OR" operations and leafs are Blake2s hashes of aggregated public keys:

- Merklized Abstract Syntax Tree
- MuSig: A New Multisignature Standard

← Serialization

Client Flow →

Enclave Architecture

The primary initial use of Trusted Execution Environments (TEE) is for enforcing payment data confidentiality (see transaction privacy), while maintaining flexibility and auditability. Other use cases may be developed in the long term.

Technology

The initial version is based on Intel SGX, but in the long-term, it would be desirable to support other TEE technology stacks, such as Arm TrustZone or RISC-V Keystone.

Transaction validation

The validation of transactions that involve payment obfuscated transaction outputs (see transaction types and accounting model) need to happen inside enclaves.

For the ease of development, the transaction validation happens in a separate process. The Chain ABCI application process then communicates with this process using a simple request-reply protocol over a 0MQ socket:

```
+-----------+ REQ +--------+
|Chain ABCI +-----+ TX val.|
|           | REP | enc.   |
+-----------+     +--------+
```

In production deployment, both of these processes should be on the same machine and hence use IPC as the underlying transport for the 0MQ messages. In development, other transport mechanisms (e.g. TCP) can be used and processes could be in different locations, for example:

- Chain ABCI is on executed on the developer laptop (any operating system), and the transaction validation enclave runs inside a Docker container (using the software-simulation mode).
- Chain ABCI is on executed on the developer laptop (any operating system), and the transaction validation enclave runs on a remote Linux machine (using the hardware mode).

Data sealing

upgrades and other enclaves, it should be sealed with MRSIGNER-derived keys.

Transaction data bootstrapping

As old payment data becomes inaccessible due to the periodic key rotation (see transaction privacy), newly joined nodes (or nodes that went offline for some time) would need a way to bootstrap the old transaction data by connecting to enclaves of remote nodes and requesting transaction data that the other nodes have locally sealed.

Lite client inside enclaves

Each enclave should internally run a lite client that would keep track of the validator set, so that it can safely store the latest “app hash” (see consensus).

Mutual attestation

The core of the bootstrapping process lies in establishing a secure channel between two enclaves – e.g. TLS, see Integrating Remote Attestation with Transport Layer Security. During the connection establishment, both parties present attestation reports that they cross-verify. This will initially utilize the Intel Attestation Service (IAS) where each full node is expected to run an IAS proxy that contains the required credentials (Key and SPID) that can be obtained on Intel portal.

In the future, the support for Data Center Attestation Primitives (DCAP) will be developed, so that each full node operator can run its own attestation service (rather than relying on IAS).

Beyond the mutual attestation, enclaves should perform additional checks against the stored app hash, e.g. if the staked state-associated with the node’s enclave is valid (or check the whitelist entry in the case of higher tier nodes).

Transaction querying

As mentioned in client flows, clients may not know their transaction data and would need to submit blind queries requesting data of some payment transactions.

For this purpose, there needs to be an enclave that can unseal the previously stored transaction data, verify the client query and return the matching transactions.

This process would again require establishing a secure connection channel between the client and the enclave (if it is remote) as in the transaction data bootstrapping – the difference is that it may only be one-side attested, as the client may not have access to the enclave architecture.

Transaction creation

When the client wishes to broadcast a payment transaction, they first need to obfuscate its content which can only be done within enclaves (that have been up-to-date with the network).

For this purpose, there needs to be an enclave that can access the current random symmetric key obtained from other enclaves (similarly to the transaction validation enclave that needs it for decryption), so that it can encrypt the payment transaction content.

Enclave breaches

If enclaves were breached, it would lead to reduced confidentiality – there would still be a level of confidentiality, as the multi-signature scheme in Chain records only limited information in the blockchain (see signature schemes). It would only affect transactions that were obfuscated with the breached key, as the key would be periodically rotated (see transaction privacy).

Note that the breach wouldn’t lead to the loss of ledger integrity, as that is preserved by the consensus algorithm.

← Client Flow

Transaction Privacy →

Transaction Privacy

Motivation

Payment data need confidentiality for many reasons, including compliance with different privacy regulations, fungibility properties etc. While confidentiality can be achieved through different means, Crypto.com Chain leverages symmetric encryption in Trusted Execution Environments for the following benefits:

- Performance due to a low overhead
- Auditability
- Flexibility in terms of what computation can be done and how the data schema can evolve

Payloads

Depending on the transaction type (see transaction types), some of its parts (transaction data, witness or both) need to be obfuscated. In that case, the broadcasted transaction binary payload includes:

- Transaction ID (if the transaction data is obfuscated)
- List of transaction inputs (if relevant) and the number of outputs (if relevant)
- Symmetric encryption-related metadata (key generation, nonce / initialization vector)
- Obfuscated payload

The encryption inside enclaves (see enclave architecture) is done using “authenticated encryption with associated data” (AEAD) scheme – the initial planned algorithm is AEAD_AES_128_GCM_SIV.

Periodic key generation

The active set of validators is involved in periodic generation of new keys that are then used by all full node enclaves on the network – the key distribution is done over mutually attested secure channels (see enclave architecture).

← Enclave Architecture

Staking →

Staking

This doc cointains the specification of the initial staking, slashing, fees and rewards mechanisms.

Account

Account is a data structure that holds state about staking:

```
pub struct Account {
    pub nonce: usize, // transaction counter
    pub bonded: Coin, // bonded staked amount
    pub unbonded: Coin, // unbonded staked amount
    pub unbonded_from: Timespec, // when the unboded staked amount can be withdrawn
    pub address: RedeemAddress, // ETH-style address; TODO: extended address?
    pub jailed_until: Option<Timespec>, // has the account been jailed in slashing-rel
    pub slashed: Option<SlashingPeriod>, // how much is the account supposed to be sla
}
```

TODO: should it have a minimum bonded amount?

Council Node

Council Node is a data structure that holds state about a node responsible for transaction validation and service node whitelist management:

```
pub struct CouncilNode {
    pub staking_address: RedeemAddress, // account with the required staked amount
    pub consensus_pubkey: crypto.PubKey, // Tendermint consensus validator-associated
    pub council_pubkey: PublicKey, // key for council node-related transaction types
    pub service_whitelist_pubkey: PublicKey, // key for service node whitelist-related
    pub nonce: usize, // update counter
}
```

Service Node

Service Node is a data structure that holds state about a high responsibility node and its offered service:

```
    pub staking_address: RedeemAddress, // account with the required staked amount
    pub service_pubkey: PublicKey, // key for service node whitelist entry updates
    pub service_type: ServiceNodeType, // what service this node offers
    pub service_url: URL, // HTTPS endpoint of the provided service
    pub nonce: usize, // update counter
}


pub enum ServiceNodeType {
    CustomerAcquirer, // (optional) custodial multi-currency wallet provider
    MerchantAcquirer, // (optional) verified merchant ID management provider, payment
    SettlementAgent, // provides currency exchange service (e.g. via IBC and DEX)
}
```

As most of the functionality is meant to faciliate some actions outside Chain, the information is recorded in the flexible form of storing a URL pointing to HTTPS API. These API endpoints will either directly execute some action or return information required for the action to be executed (e.g. other protocol's details).

Transaction Fee

The minimal transaction fee is defined according to the formula:

```
<BASE_AMOUNT> + <PER_BYTE> * size
```

`BASE_AMOUNT` and `PER_BYTE` are special network parameters in a fraction of CRO. `size` is the serialized transaction data's size in bytes.

Basic (`TransferTX` , `DepositStakeTx` , `WithdrawUnbondedTx` , `UnbondStakeTx` ,) transaction types need to check

```
sum(inputs amounts) or account.unbonded/bonded == sum(outputs amounts) + fee
```

The fee goes to the rewards pool.

For Advanced TX types (council node and service node state metadata management), the initial prototype will not require a fee.

Rewards

```
pub struct RewardsPool {
    pub nonce: usize, // update counter
    pub total_remaining: Coin, // total available amount
    pub last_update: Timespec, // when was the pool last updated
}
```

The initial prototype will have a `DAILY_DISTRIBUTION_AMOUNT` network parameter; later on, it should be a function that takes a target emission rate, total remaining amount etc. as parameters.

Distribution

Each validator will maintain this structures (perhaps persistent / on-disk): `day_claim_council_node: Map<CouncilNode, BlockCount>`

The actual distribution then happens once in a while in BeginBlock, here's a pseudo code:

```
for each validator in BeginBlock.LastCommitInfo:
    lookup validator's staking_address
    if day_claim_council_node[validator_day_claim_council_node] is Nil:
        day_claim_council_node[validator_day_claim_council_node] = 0
    day_claim_council_node[validator_day_claim_council_node] += 1

for each node in day_claim_council_node:
    if node.staking_address.jailed_until.is_some() OR node.staking_address.jailed_unti
    node.slashed.is_some() OR
    node.staking_address.bonded < COUNCIL_NODE_MIN_STAKE:
        remove node from claim set

if BeginBlock.time >= RewardsPool.last_update + 24 hours:
    validator_amount = min(DAILY_DISTRIBUTION_AMOUNT, RewardsPool.total_remaining)

    for each node in day_claim_council_node:
        node_share = compute_share(node, day_claim_council_node, validator_amount)
        END/COMMIT_BLOCK_STATE_UPDATE(node.staking_address.bonded += node_share; node.

    END/COMMIT_BLOCK_STATE_UPDATE(RewardsPool.total_remaining -= amount; RewardsPool.n
    day_claim_council_node = {}
```

`compute_share` determines the amount for each claiming node; the initial prototype will compute it:

- council nodes: proportional to the signed block count (node block signatures / all block validators signatures)

> The draft technical whitepaper says there will be rewards for acquirer nodes. These terms may be revised in later drafts.

Slashing

This part describes functionality that aims to disincentivize network-observable actions, such as faulty validations, of participants with values at stake by penalizing/slashing them. The penalties may include losing some amount of their stake (surrendered to the rewards pool), losing their ability to perform the network functionality for a period of time, collect rewards etc.

Slashing period

Some of the faulty behavior may be non-malicious, e.g. due to misconfiguration. In order to mitigate the impact of these initially likely categories of faults, there is a concept of a slashing period whereby the penalty is capped. As penalties are specified in fixed decimal fractions of a stake, it is the highest fraction. In other words, in a given period, a violator is punished for the worst violation rather than losing all stake.

Even with this “more tolerant” punishment setup, it will still be quite expensive and desirable to avoid. Note that this setup must be accompanied by immediate jailing / limiting the violater’s network ability.

This is a sketched out additional data structure to keep track of (a part of Account):

```
pub struct SlashingPeriod {
    pub start: Timespec, // when the period started,
    pub end: Timespec, // when the period ended,
    pub slashed_ratio: Decimal, // fixed decimal; cumulative so far slashed fraction;
}
```

Unjail

When the previous violator wishes to resume the functionality, the node operator can send a signed `UnjailTx` . Its validation logic is the following:

```
block_time = (... from beginblock ...)
account = (...lookup / recover from signature...)
if account not found:
    return invalid("account not found")
    return invalid("account not jailed")

if nonce != account.nonce + 1:
    return invalid("invalid nonce")

if block_time < account.jailed_until.unwrap() || (account.slashed.is_some() && account
    return invalid("account still jailed")

if !check_signature(txid):
    return invalid("invalid signature")

END/COMMIT_BLOCK_STATE_UPDATE(deduct(account); account.jailed = None)
```

TODO: auto-unjailing?

Tracking

Validators / Council Nodes

Each BeginBlock contains two fields that will determine penalties:

- LastCommitInfo: this contains information which validators signed the last block
- ByzantineValidators: evidence of validators that acted maliciously

Their processing is the following:

```
for each evidence in ByzantineValidators:
    if evidence.Timestamp >= BeginBlock.Timestamp - MAX_EVIDENCE_AGE:
        account = (... get corresponding account ...)
        if account.slashing_period.end is set and it's before BeginBlock.Timestamp:
            account.slashing_period.slashed_ratio = max(account.slashing_period.slashe
        else:
            if account.slashing_period.end is set and it's after BeginBlock.Timestamp:
                END/COMMIT_BLOCK_STATE_UPDATE(deduct(slashing_period, account))

            account.slashing_period = Some(SlashingPeriod(start = Some(BeginBlock.Time
                end = Some(BeginBlock.Timestamp + SLASHING_PERIOD_DURATION, slashed_ra
            account.jailed_until = Some(BeginBlock.Timestamp + JAIL_DURATION)


for each signer in LastCommitInfo:
    council_node = (... lookup by signer / tendermint validator key ...)
    update(council_node, BLOCK_SIGNING_WINDOW)

for each council_node:
    missed_blocks = get_missed_signed_blocks(council_node)
```

```
if account.slashing_period.end is set and it's before BeginBlock.Timestamp:
    account.slashing_period.slashed_ratio = max(account.slashing_period.slashe
else:
    if account.slashing_period.end is set and it's after BeginBlock.Timestamp:
        END/COMMIT_BLOCK_STATE_UPDATE(deduct(slashing_period, account))

    account.slashing_period = Some(SlashingPeriod(start = Some(BeginBlock.Time
        end = Some(BeginBlock.Timestamp + SLASHING_PERIOD_DURATION, slashed_ra
    account.jailed_until = Some(BeginBlock.Timestamp + JAIL_DURATION)
```

“Global state” / APP_HASH

Tendermint expects a single compact value, `APP_HASH` , after each BlockCommit that represents the state of the application. In the early Chain prototype, this was constructed as a root of a Merkle tree from IDs of valid transactions.

In this staking scenario, some form of “chimeric ledger” needs to be employed, as staking-related functionality is represented with accounts. In Eth, Merkle Patricia Trees are used: https://github.com/ethereum/wiki/wiki/Design-Rationale#merkle-patricia-trees (The alternative in TM: https://github.com/tendermint/iavl depends on the order of transactions though)

They could possibly be used to represent an UTXO set: https://medium.com/codechain/codechains-merkleized-utxo-set-c76c9376fd4f

The overall “global state” then consists of the following items:

- UTXO set
- Account
- RewardsPool
- CouncilNode
- ServiceNode
- MerchantWhitelist (not yet spec-out / TODO)

So each component could possibly be represented as MPT and these MPTs would then be combined together to form a single `APP_HASH` .

Network Parameters

This section aims to collect all the mentioned network parameters:

- `BASE_AMOUNT`
- `PER_BYTE`

- `PER_MERCHANT_MIN_STAKE`
- `CUSTOMER_ACQUIRER_NODE_MIN_STAKE`
- `COUNCIL_NODE_MIN_STAKE`
- `MAX_EVIDENCE_AGE` (Note that currently in EvidenceParams of ABCI, there's MaxAge set in the number of blocks, but here we assume time)
- `SLASHING_PERIOD_DURATION`
- `JAIL_DURATION`
- `BLOCK_SIGNING_WINDOW`
- `MISSED_BLOCK_THRESHOLD`
- `BYZANTINE_SLASH_RATIO`
- `LIVENESS_SLASH_RATIO`

TODO: TX that can change them?

END/COMMIT_BLOCK_STATE_UPDATE

Besides committing all the relevant changes and computing the resulting `APP_HASH` in BlockCommit; for all changes in Accounts, the implementation needs to signal `ValidatorUpdate` in EndBlock if the change is relevant to the council node's staking address:

- if the `bonded` amount changes and >= `COUNCIL_NODE_MIN_STAKE` , then the validator's power should be set to that amount
- if the `bonded` amount changes and < `COUNCIL_NODE_MIN_STAKE` , then the validator's power should be set 0
- if the `jailed_until` changes to `Some(...)` , then the validator's power should be set 0
- if the `jailed_until` changes to `None` (unjailtx or auto?) and `bonded` amount >= `COUNCIL_NODE_MIN_STAKE` , then the validator's power should be set to the `bonded` amount

InitChain

The initial prototype's configuration will need to contain the following elements:

- Above network parameters
- ERC20 snapshot state / holder mapping `initial_holders` : `Vec<(address: RedeemAddress, is_contract: bool, amount: Coin)>` (or `Map<RedeemAddress, (bool, Coin)>`)
- Network long-term incentive address: `nlt_address`
- Secondary distribution and launch incentive addresses: `dist_address1` , `dist_address2`
- initial validators: `Vec<CouncilNode>`
- bootstrap nodes / initially bonded: `Vec<RedeemAddress>`

The validation of the configuration is the following:

2. check that sum of amounts in `initial_holders` == MAX_COIN (network constant)
3. check there are no duplicate addresses (if Vec) in `initial_holders`
4. for each `council_node` in `Vec<CouncilNode>` : check `staking_address` is in `initial_holders` and the amount >= `COUNCIL_NODE_MIN_STAKE`
5. for each `address` in initially bonded `Vec<RedeemAddress>` : check `address` is `initial_holders` and the amount >= `COMMUNITY_NODE_MIN_STAKE`
6. size(Validators in InitChainRequest) == size(`Vec<CouncilNode>`) and for every CouncilNode, its `consensus_pubkey` appears in Validators and power in Validators corresponds to staking address' amount

If valid, the genesis state is then initialized as follows:

1. initialize RewardsPool's amount with amounts corresponding to: `nlt_address` + `dist_address1` + `dist_address2`
2. for every council node, create a `CouncilNode` structure
3. for every council node's staking address and every address in initially bonded: create `Account` where all of the corresponding amount is set as `bonded`
4. for every address in `initial_holders` except for `nlt_address` , `dist_address1` , `dist_address2` , council nodes' staking addresses, initially bonded addresses: if `is_contract` then add the amount to RewardsPool else create `Account` where the amount is all in `unbonded` and `unbonded_from` is set to genesis block's time (in InitChain)

← Transaction Privacy

Notes on Production Deployment →

Client Flow

Block-level filtering

The full node “tags” each block with a probabilistic filter (tagging is done using the Events in Tendermint). This filter is then used by the light client code to determine which blocks are of any interest, requests these blocks and processes them. This current mechanism is similar to BIP 157.

Currently, the used filter is the Bloom filter used in Ethereum defined with the following parameters: m = 2048 (bits; keccak-256 hash function), k = 3 (yellowpaper page 5)

The full node inserts the following data into the filter at the moment:

1. “view keys” (secp256k1 public keys allowed to view the content) in the case of transactions producing UTXOs (see transaction types and transaction privacy mechanism for more details).
2. Staked state addresses in the case of transactions manipulating accounts (see accounting details).

Client knows transaction

Each block header includes an application hash (“APP_HASH”, see consensus for details). As a part of it is a root of a Merkle tree of valid transactions, a client can check whether its known transaction was included in a block:

1. Get the block’s application hash / header information.
2. Compute the transaction ID from transaction data (see transaction).
3. Check a Merkle inclusion proof where the transaction ID is one of the leaves, and a part of the application hash is the root.

Client doesn’t know transaction data

If the client doesn’t know transaction data, it can collect valid transaction identifiers from blocks that matched its data using the block-level filter. If the transaction data was transparent (staked state operations), the client can decode transaction directly by requesting the full block data. If the transaction data was obfuscated (payments), the client needs to contact an enclave and prove they can access transaction data using view key signatures (see transaction privacy and enclave architecture for more details).

running their own full node) etc.

Payment transaction submission

When the client wishes to submit a payment transaction (UTXO-based), they will construct a plain signed transaction and submit it to one of full node’s enclaves over a secure channel (see enclave architecture).

← Signature Schemes

Enclave Architecture →

Notes on Performance

The current discourse in cryptocurrencies focuses on “maximum TPS” as the key defining performance metric and one often finds various outlandish claims about it. For more details why these numbers don’t matter, we recommend this article from the Nervos Network.

> **NOTE**
>
> There are more issues with regards to “maximum TPS metrics” that are not mentioned in the article

We’ll have to eat humble pie and say that some of the existing Crypto.com Chain materials fall into similar fallacies (despite being written with the word “aim” and intended to be seen in the potential Layer-2 context).

In the future, once the system reaches certain maturity, we’d like to establish metrics (e.g. the 95th percentile of transaction end-to-end latency) that are more meaningful for user experience. We would try to benchmark against these metrics, ideally in diverse settings (rather than “the best case where nothing ever crashes”) and using estimated real workloads.

← Notes on Production Deployment

Threat Model →

Notes on Production Deployment

- See Tendermint notes on running in production
- Validators shouldn’t be exposed directly to the internet
- RPC shouldn’t be exposed directly to the internet (as it currently doesn’t support rate-limiting, authentication…)
- Validator block signing should be via kms

← Staking

Notes on Performance →

Threat Model

Threat modeling is a systematic approach to find potential threats by decomposing and enumerating system components. There are many different methodologies and/or frameworks when conducting threat modeling, such as STRIDE, DREAD, Attack Tree, etc. In our case, our Thread Model is based on STRIDE and Attack Tree.

STRIDE provides a set of security threats in five categories (the last one is ignored):

1. Spoofing - impersonating the identity of another
2. Tampering - data is changed by an attacker
3. Repudiation - an attacker refuses to confirm an action was conducted
4. Information Disclosure - exposing sensitive information
5. Denial of Service - degrade the availability or performance of the system
6. Elevation of Privilege (not applicable in our case)

For each category, we enumerate all the potential threats by breaking down a high-level goal into more specific sub-goals, in a way similar to attack tree enumeration. And in each sub-goal, we set the risk level by combining the `Severity` and `Exploitability` of the item.

Severity

- `5` : Severe impact on the whole system
- `4` : High impact on the whole system
- `3` : Moderate impact on the whole system or Severe impact on individual user/node
- `2` : High impact on individual user/node
- `1` : Moderate impact on individual user/node

Exploitability

- `5` : Existing exploit code available
- `4` : Relatively easy to exploit
- `3` : Attack is practical but not easy, a successful attack may require some special conditions
- `2` : Theoretically possible, but difficult in practice
- `1` : Very difficult to exploit
- `0.1` : Almost impossible

Assets

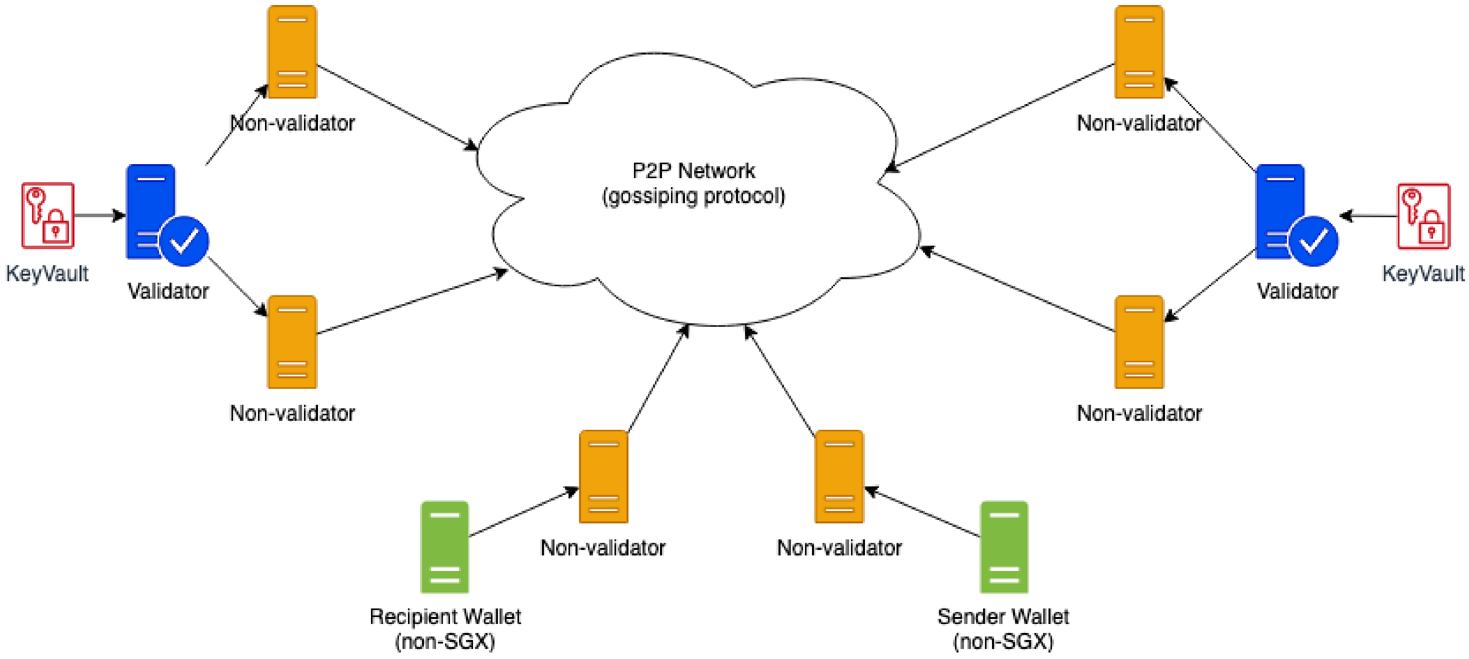
1. **The integrity of the account balance**: the most important piece of information in the blockchain.
2. **Validator secret keys**: one of the most powerful key, lost 1/3 of these keys will render the whole system to an unstable state.
3. **User secret keys**: key owner implies fund owner
4. **Transaction encryption keys**: transaction privacy of the system relies on the secrecy of this key

Scope

The whole Crypto.com Chain is a complex system and involves many different components. And therefore, the scope of this threat model is limited only to the major components of the system. To be more specific, the threat modeling of Tendermint and Intel SGX is **NOT** in the scope of this threat modeling.

We also assume standard security measures like OS level hardening, software patching, anti-virus, network firewall, physical security, etc, are properly implemented, executed and monitored. These mitigation strategies are not mentioned here.

System Diagram



Result

HERE

← Notes on Performance