REDACTED

Limited Review of Java-Tron

August 26, 2024

Produced for

TRON

by

CHAINSECURITY

Contents

1 Executive Summary 3
2 Assessment Overview 5
3 Limitations and use of report 17
4 Terminology 18
5 Findings 19
6 Resolved Findings 25
7 Informational 36
8 Notes 40

1 Executive Summary

Dear Tron-Team,

Thank you for trusting us to help you with this code review. Our executive summary provides an overview of subjects covered in our review of Java-Tron according to Scope to support you in forming an opinion on their security risks.

Tron uses Java-Tron as the node software to run the Tron network. Hence, Java-Tron is (among other things) responsible for executing transactions, generating blocks, achieving consensus and operating the peer-to-peer network.

Due to the complexity of Java-Tron and the limited allocated time, this review cannot uncover all the bugs inside of it. Instead, the goal of this review was to uncover as many bugs as possible while focusing on the following parts of the code:

- Tron Virtual Machine (TVM)
- Consensus
- Peer-to-Peer (P2P)

Some of the most significant findings are:

- PBFT Messages Create State Expansion
- Unpermissioned Censoring of Fork Blocks
- Resource Consumption by Blocks Not Signed by Witnesses

These three findings have all been addressed through code corrections. For some other issues, the risks have been accepted based on the assumption of economically acting super representatives. Lastly, some issues with non-critical severity have been redacted to prevent malicious actors from creating disturbances.

It is important to note that such reviews are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	4
• Code Corrected	4
Medium-Severity Findings	8
• Code Corrected	4
• Specification Changed	1
• Risk Accepted	3
Low-Severity Findings	9
• Code Corrected	5
• Risk Accepted	4

Please note that upon request of the customer certain findings have been redacted from this report.

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Java-Tron repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

Java Tron

V	Date	Commit Hash	Note
1	December 11, 2023	440d062dc7d39eff532414cc3b887f61509ef9f5	v4.7.3
2	May 30, 2024	a8ad2a169e58946b5b8de6ecf7b7ef5b8db05aff	v4.7.5

Protocol

V	Date	Commit Hash	Note
1	December 11, 2023	961d0632b67971762dc5a8eefab50ed08d5c8673	v4.7.3

The review is based on Oracle JDK 1.8.

Java-Tron has a very large codebase. In order to concentrate efforts, the focus of this review was on the following three packages of the code:

- Tron Virtual Machine (TVM)
- Consensus
- Peer-to-Peer (P2P)

However, due to their individual complexity, these packages were not covered in their entirety.

2.1.1 Excluded from scope

Generally, any part of the codebase not mentioned above is out-of-scope. More explicitly, features like shielded transactions, database performance and cryptographic implementations were out of scope. Furthermore, additional components of the Java-Tron ecosystem were out of scope. Among others, such projects include the modified solidity compiler and mechanisms for snapshot synchronization.

2.2 System Overview

This system overview describes the initially received version (**Version 1**) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

2.2.1 Tron Execution Layer

2.2.1.1 Transactions

The Tron execution layer processes transactions. There are multiple types of transactions in Tron, the most common types being the contract creation and contract call transactions. Other transaction types are Tron-specific and are used to interact with the chain, such as freezing TRX, voting for witnesses, or interacting with TRC-10 tokens.

The entry point of the execution layer is the `Manager`. When a transaction needs to be processed, the function `processTransaction()` is called. There are multiple contexts where it can be called:

- Upon receiving a new block, all of the block's transactions are processed sequentially, updating the state accordingly.
- When the client is a witness and it is their turn to produce a block, transactions are processed to create the block. In that case, state changes are discarded after the block is produced and will be reprocessed when the block is received upon broadcasting it.
- Transactions received from other nodes or by users via RPC calls are processed without applying the corresponding state change immediately. If they were sent to be included in a block, they are added to the pool of pending transactions.

`Manager.processTransaction()` has multiple responsibilities. First, several checks are performed on the transaction:

- The signature's referenced block number and hash are checked to ensure that the transaction was meant to be submitted on the given chain and fork.
- The transaction is checked not to be expired.
- The size of the transaction is checked to be smaller than or equal to the maximum allowed size.
- The transaction is checked to ensure that it is not a duplicate. This is done by checking if the transaction's ID (hash) is already in the state's transaction history.

Then, transaction-wide fees are consumed, including:

- The bandwidth of the transaction.
- The MultiSig fee, if the transaction is a multi-signature transaction.
- The Memo fee, if the transaction has a memo.

After that, the transaction is passed over `RuntimeImpl.execute`. This function is responsible for finding the actuator that corresponds to the transaction type. Each transaction type is supported by the client via an actuator. In the general case, there is exactly one actuator for a given transaction type. The only exception is for smart contract transactions, where the `VMActuator` is responsible for both calls and the creation of smart contracts, which in Tron are two different transaction types, as opposed to Ethereum where the creation of smart contracts is done with a call with an empty `to` field.

An actuator defines two main methods:

- `validate()`: This method is called to perform several checks on the transaction, from ensuring that the transaction's parameters are valid to checking if the sender has enough balance in case the transaction is a transfer.
- `execute()`: This method is called to execute the transaction. It is responsible for updating the state of the chain with the effect of the transaction.

Hence, at this point, the `RuntimeImpl` first calls the actuator's `validate()` method and then its `execute()` method.

In the case of the `VMActuator`, the `execute()` method creates a program given the bytecode of the contract being called or the init-bytecode to execute in the case of contract creation and calls

`VM.play()` to run the TVM against the program. For other actuators, the VM is not involved, and the actuator is responsible for updating the state of the chain directly.

Once the transaction has been executed, the Manager performs several post-execution actions such as deleting the contracts that have suicided during the execution.

2.2.1.2 The Tron Virtual Machine (TVM)

2.2.1.2.1 The Repository

The Repository is the entry point to read and write the state of the chain from within the TVM. It abstracts away database operations and cache management. Examples of states being often read or written are accounts, chain parameters or smart contract codes. The Repository implements a commit/discard mechanism to allow for discarding changes made in case of a reverting transaction, failed external call or client exception such as a stack overflow. The function `newRepositoryChild` can be called to get a new context where changes can be committed or discarded independently of the parent context. Similarly, the function `commit` is used to commit the changes made in the given repository. If the repository is a child, the changes are committed to the parent repository. If the repository is the root, the changes are committed to the relevant stores directly. It should be noted that any state read and changes should hence be made through the repository to ensure that the changes are correctly committed or discarded.

2.2.1.2.2 The Program

A program can be seen as an object that can be executed by the TVM. It contains not only the bytecode to be executed, but also a `Repository`: the state to use for the execution, and some TVM-related context such as the caller, the stack, the memory, the context address, the code address or the program counter.

A new program is created in the following cases:

- When a smart contract call transaction is processed, the `VMActuator` creates a program from the account and code of the called contract.
- When a contract creation transaction is processed, the `VMActuator` creates a program from the newly created account and the init-bytecode supplied in the transaction.
- When the TVM executes a `CREATE` or `CREATE2` opcode, a new program is created for the new execution context used to execute the init-bytecode.
- When the TVM executes a call opcode (`CALL`, `DELEGATECODE`...), a new program is created for the new execution context.

Two metrics are used to limit the execution of a program:

- The energy: Any given program is given an amount of energy to execute. The energy is consumed by the TVM as the program is executed. Each opcode consumes a certain amount of energy. If the energy is depleted before the program is executed fully, the execution is stopped with an `OutOfEnergyException`.
- The time: A big difference with the EVM is that in Tron, transactions are given a CPU time limit to execute. If the time limit is reached before the program is fully executed (or before the energy is depleted), the execution is stopped with a `OutOfTimeException`.

2.2.1.2.3 The Jump Table

The Jump Table maps opcodes to tuples of energy function, execution function, required stack arguments and produced stack arguments. The energy function takes as input a program and returns the amount of energy consumed by the opcode given the program's state. The execution function also takes as input a program and updates the program state accordingly. The required and produced stack arguments are used to check that the stack is in the correct state before and after the execution of the opcode.

There are different Jump tables for different hardforks of the chain, as when new opcodes are added modified, or removed, the jump table is updated accordingly. The class `OperationRegistry` keeps track of the different jump tables as new Tron versions are released.

2.2.1.2.4 Execution

A program is executed by the TVM given a jump table via the function `VM.play`. The function iterates over the program's bytecode and processes the opcodes one by one. For each opcode of the bytecode, the function fetches the relevant tuple from the jump table. It is then ensured that there are enough stack arguments to execute the opcode and that the execution of the opcode will not overflow the stack. Afterward, the energy consumed by the opcode is computed and subtracted from the program's energy. Shall the energy be depleted, the execution is stopped with an `OutOfEnergyException`. If the execution time of the program exceeds the time limit, the execution is stopped with an `OutOfTimeException`. Finally, the opcode is executed, its effects are applied to the program and the produced stack arguments are pushed onto the stack.

Most opcodes are simple and only require a few lines of code to be executed. However, some opcodes require more complex logic.

- calls, `CREATE` and `CREATE2` opcodes are responsible for creating a new program to execute the called contract/init bytecode and handling the return value of the program.
- Several opcodes specific to the TVM such as `DELEGATERESOURCE`, `FREEZEBALANCEV2` or `VOTEWITNESS` are handled in processors that are responsible for the specific logic of the opcode. These processors are similar to the corresponding actuators that are used to process corresponding instructions when given in the form of transactions. For example, the `FreezeBalanceV2Processor` and the `FreezeBalanceV2Actuator` have similar logic to handle the freezing of TRX.

2.2.1.2.5 Differences with the EVM

The TVM is different from the EVM in multiple aspects, we give here a non-exhaustive list of differences:

- The TVM semantics do not match any Ethereum hardfork, rather several EIPs are implemented in the TVM but not all of them.
- Several opcodes have different semantics, for example `DIFFICULTY` and `GASLIMIT` return zero in the TVM.
- The energy has a fixed price in SUN where the gas price is variable in the EVM.
- Several gas prices are different than in the EVM. For example, there is no concept of the access lists in the TVM.
- Because of the `0x41` prefix, the `CREATE2` opcode computes the to-be-created contract address differently than in the EVM.
- Several precompiles behave differently or are accessible using different addresses than in the EVM.
- The TVM adds TRC-10 related opcodes `CALLTOKEN`, `TOKENBALANCE`, `CALLTOKENVALUE` and `CALLTOKENID`, which are used to interact with TRC-10 tokens. A program hence, in addition to having a `callvalue`, can have a `calltokenvalue` and a `calltokenid`. As opposed to `callvalue`, `calltokenvalue` and `calltokenid` of the given context are not forwarded to a sub-context created using a `DELEGATECALL`.

- The opcode `ISCONTRACT` is used to check if an address is a contract or an account.
- Batch validations for normal and multiple signatures.
- Anonymous contract and Librustzcash-related pre-compile contracts
- The `CREATE` opcode does not behave the same as in Ethereum as the transaction hash is used together with the nonce of the contract to compute the new address. The nonce of contracts is incremented by one after each call or contract creation as opposed to each contract creation for the EVM. Additionally, the nonce of a contract is not persistent between transactions.
- Stake 1.0 related opcodes: `FREEZE(0xd5)`, `UNFREEZE(0xd6)` and `FREEZEEXPIRETIME(0xd7)`
- Contract voting-related opcodes and pre-compiled contracts
- Stake 2.0 related Freeze / Unfreeze / Delegate / UnDelegate opcodes and pre-compiled contracts.
- If a contract tries to send TRX to itself, this will fail.
- TVM has an explicit memory size limit, whereas EVM just has quadratic cost.

2.2.2 Tron Consensus

2.2.2.1 Resource model

Tron offers a transaction anti-spam system based on the acquisition and spending of two resources: Energy and Bandwidth.

Every transaction requires the consumption of an amount of *bandwidth* equal to the size in bytes for the serialized transaction. Every account has the right to 600 units of free bandwidth per day. A further amount of daily bandwidth allowance can be acquired by staking TRX, the native token of the Tron blockchain, or bandwidth for a transaction can be acquired by paying a transaction fee in TRX, which gets burned. When paying the transaction fee, bandwidth is priced at 0.001 TRX per byte.

The other resource used when pricing transactions is *Energy*. Energy can be seen as an equivalent to *gas* in Ethereum. Energy is spent in transactions of type `TriggerSmartContract` (contract calls) and `CreateSmartContract` (smart contract creations where the constructor code is also executed). Energy can be acquired by staking TRX, or by paying a transaction fee for Energy, at a price of 420 *sun* (10^-6 TRX) per unit of energy. As in the case of *bandwidth*, the fee is burned.

Finally, some transactions require the spending of a TRX fee when executed. These are `AccountPermissionUpdateContract`, `AssetIssueContract`, `AccountCreateContract`, `ExchangeCreateContract`, `MarketCancelOrderContract`, `MarketSellAssetContract`, `WitnessCreateContract`.

Extra fees are also applied if the transaction has a multisignature (0.001 TRX), and if the transaction has a *memo*, i.e. the `data` field of the transaction protobuf is not empty (0.01 TRX).

Staking allows the acquisition of self-renewing resources for daily usage. Users can stake TRX in exchange of either *bandwidth* or *energy*. A daily amount of $43.2*10^9$ bandwidth units (bytes) are available to stakers, and $90*10^9$ *energy* per day. When a user stakes, one of these two resources is acquired, so that it can be spent. The global daily amount is proportionally partitioned among the stakers.

Acquired resources can be spent, and the amount used becomes available again linearly of the course of 24 hours, i.e. after 12 hours, it has regenerated by 50%, after 24 by 100%.

2.2.2.2 Staking

Tron has two versions of staking. The first was applied through transaction type `FreezeBalanceContract`, and is referred to as *staking v1*, it is now deprecated and new stakes cannot be created. The new version is applied through transaction type `FreezeBalanceV2Contract`, or opcode `FREEZEBALANCEV2` (`0xda`) if the staker is a smart contract, and is referred to as *staking v2*. The user specifies the amount to be frozen, and the type of resource to be acquired (*bandwidth* or *energy*). Internal balances are kept in the account (`frozenV2` field in the `account` protobuf), for the TRX amount staked for bandwidth and energy. Staking operations increment the global values `TOTAL_ENERGY_WEIGHT`, and `TOTAL_NET_WEIGHT`, so that the total available daily bandwidth and energy can be attributed proportionally to users stakes.

Staking v1 required the user to lock the stake for 3 days, after which the user could withdraw the staked TRX at any time. Stake v2 has no lock duration, however when unstaking the acquired resource in unallocated from the user, the voting power is lost, and the user has to wait for an unfreeze delay of 2 weeks before reacquiring the TRX being unstaked.

2.2.2.3 Voting power

Staking for bandwidth or energy automatically entitles the staker to another kind of resource: voting power. The voting power of an account is the sum of the frozen balances for energy and bandwidth, both for v1 and v2 staking.

2.2.2.4 Delegations

Resources (*bandwidth* and *energy*) can be delegated by stakers to other accounts, such that resource marketplaces can be created. Two versions of resource delegations exist: v1, linked to staking v1, is historical and deprecated, no new delegations can be created. However, some still exist. V2 delegations are the ones in current use, linked to staking v2.

When delegating to an account, the delegator transfers part of their unused resources to the delegate through a `DelegateResourceContract` type of transaction, or through the `DELEGATERESOURCE` opcode (`0xde`), if the delegator is a smart contract. The delegator can specify whether to lock the delegated resource for a period of time, which means they can't be reacquired until the lock has expired. When delegating resources, the voting power attached to them is maintained by the delegator.

2.2.2.5 Voting

The Tron consensus protocol is denominated Delegated Proof-of-Stake (*DPoS*). In Tron DPoS, 27 witnesses are elected every consensus cycle (6 hours) by stakers of TRX. The elected witnesses (Super Representatives) take turns (one per block) in performing the essential blockchain activity of block production. Stakers are incentivized to participate in the election of Super Representatives because block rewards are distributed back to the voters. A staker can participate in voting with the transaction type `VoteWitnessContract`, or with opcode `VOTEWITNESS` (`0xd8`), if the voter is a smart contract. A voter can distribute their vote towards up to 30 witness candidates. New votes from a voter overwrite their past votes. Votes are counted at the end of the 6-hour consensus period called cycle. At the end of each cycle, votes are counted and accumulated into the witness accounts. The top 27 witness candidates become *super representatives*. The next 100 witness candidates are elected *standby representatives* but these do not have specific tasks towards maintenance of consensus, however, they are allocated a share of the block reward.

2.2.2.6 Proposals

Witnesses are special accounts that participate in the consensus protocol at the peer-to-peer level however they also participate in the protocol by sending transactions to create a new proposal (`ProposalCreateContract`) and voting for proposals (`ProposalApproveContract`). Proposals can modify the blockchain parameters defined in java class `ProposalUtil`. The parameters include fees, global available energy and bandwidth, the acceptance by the blockchain of new transaction types and mechanisms.

Once a proposal is created, it is visible to all blockchain participants. After 3 days, at the end of a cycle, just before the maintenance period, if it has been approved by a super-majority of the current representatives (18 out of 27), the proposal is approved. Otherwise, it is not approved.

2.2.2.7 Rewards

Every block produced generates rewards towards stakers and witnesses. The first 127 witnesses (the standby witness set) share a common block reward of 160 TRX. These are distributed to the witnesses proportionally to their shares of total votes. Every witness sets a *brokerage ratio*, which is the share of their rewards that they get to keep. The rest goes toward their voters.

The distribution algorithm of the witness reward to the voters is similar to Synthetix rewards algorithm, aka Masterchef. It guarantees an O(1) execution time when a voter withdraws rewards, regardless of the time elapsed since the last withdrawal and the number of voters.

Each block also attributes 16 TRX to the witness who produces it, which likewise gets split between the witness (brokerage ratio), and its voters.

2.2.2.8 Block Generation

Blocks are generated by full nodes who are active witnesses (Super Representatives). The Java class `DposTask` runs on a separate thread that checks at every new slot time whether the full node should generate a block. The 27 Super Representatives for a cycle take turns to propose blocks in a round-robin way.

On block generation, queued transactions are packed into the new block. Transactions are processed by executing their payload on the current state. Every transaction builds on the state of previous transactions. The time to generate a block is capped by default to a quarter of the block period (quarter of 3 seconds) so that validating nodes will have time to process the new block before it is time for the production of a subsequent block.

2.2.2.9 Fork choice

Tron DPoS consensus uses a fork choice rule of longest chain. If a fork happens, and a block is received which has the same block number as a previously processed block, a fork is detected, and the new block is saved in the Khaos database for block storage if a parent is found in the database. The new block is not yet used. When a block whose block number is higher than the current head block, but whose parent is not the current head, is received, it is checked that its parent resides in the Khaos database (it is known to the full node, and so recursively for ancestors). In this case, the new chain is longer than the currently used one, and a reorganization can happen. The chain is reverted, one block at a time, up to the height where the fork happened. Then, the blocks from the new chain are one by one applied.

2.2.2.10 Finality

Blocks are considered final when 19 super representatives confirm them. When producing a new block, a super representative implicitly gives confirmation for all ancestor blocks. Hence, finality is implied by the block being used as ancestor in a chain where at least 18 other super representatives have contributed subsequent blocks. A finalized block is called a "solidified" block in Tron jargon. The "solid" block of the chain is the latest solidified block. Events are emitted when a block is solidified so that users can subscribe to a solidified feed and receive definitive confirmation of on-chain events.

2.2.3 Tron P2P messaging

The Tron P2P layer is initialized by the `TronNetService` which is the central object that initializes all other related objects such as services, checks and handlers. Note that it sets up the `P2PService`, an object from the Tron p2plib library that manages the underlying channels and connections and is responsible for receiving and sending messages. Further, more services are started that manage synchronization, advertising and more.

The `P2PService` offers the possibility to register handlers that are called upon for the reception of messages of non-negative types. Namely, the handler registered corresponds to the `P2pEventHandlerImpl`, which, as a consequence of compatibility with the `P2PService`, implements the three hooks:

1. `onConnect`
2. `onDisconnect`
3. `onMessage`

`onMessage` will be the hook called when messages are received and will essentially forward the messages to more specialized message handlers suitable for the type of message received.

Note that several thread executions will run (typically each service will start a thread periodically) to process data while the `P2PService` will run one scheduler thread that will distribute workloads on worker threads. Hence, the `P2pEventHandlerImpl`'s hooks may run concurrently. However, per peer (or channel) only one execution at a time may run due to the design of the p2plib.

2.2.3.1 Connections and Peer Management

2.2.3.1.1 Establishing Connections

As the name suggests, `onConnect` is triggered when a connection is established. Namely, that occurs once the low-level handshake of the p2plib has been completed. As a consequence, a `PeerConnection`, wrapping the channel and storing peer-related data, is constructed while the peer is tracked as a peer. A P2P-level handshake is initiated through `HandShakeService.startHandshake`. That ultimately sends a `HelloMessage`.

Note that typically, it is expected that a `HelloMessage` is received so that one can validate the compatibility of the two nodes (e.g. P2P protocol version) and potentially start a synchronization. More specifically, a synchronization process (with `syncService.startSync`) is initiated if the `HelloMessage`'s sender has a higher head block number. If not, the peer is tagged to require a sync if its head block number is lower than the one that the recipient has (see Synchronization)

2.2.3.1.2 Disconnecting

Note that nodes can disconnect for several reasons. Either node could disconnect. If a node wants to disconnect, it calls `PeerConnection.disconnect` for the particular peer due to reasons on the protocol level (e.g. bad behaviour). That shares a `DisconnectMessage` with the peer and closes the channel. Typically, nodes will disconnect on bad messages, timeouts or other errors during the processing of a peer's messages.

If the peer wants to disconnect, the disconnect message will be received and the channel will be closed. However, in case no disconnect message is received, the channel will eventually be closed due to a close future on the underlying `netty` channel.

Ultimately, once a channel has been marked as disconnected either the hook `onDisconnect` in the `P2pEventHandlerImpl` will be invoked (due to the close future on the underlying `netty` channel) or the thread run by the `PeerManager` periodically instantiated for checking for disconnects will properly untrack the peer completely.

Note that additionally, there is `PeerStatusCheck` which hosts a periodically scheduled thread that ensures that no timeouts occur for communication about blocks, inventory requests and synchronization block requests (see Synchronization and Transactions and Blocks). Timed-out nodes are disconnected.

2.2.3.1.3 Receiving Messages

This section elaborates on how incoming messages are handled. Namely, the p2plib will invoke the `onMessage` hook of the event handler implementation. The message is processed by the corresponding message handler's `processMessage` function (except the `HelloMessage` which is processed by a `processHelloMessage`) function. Note that any exception raised will ultimately lead to a disconnection with a ban time of typically 60 seconds (as defined in the p2plib). However, in case of bad transactions, the ban time will be one hour. The sections below will put the message processing into the right context.

2.2.3.2 Synchronization

2.2.3.2.1 Initiating Synchronization

The synchronization process is started when

1. a `HelloMessage` is received and the peer claims to know more (see Establishing Connections)
2. or a block of which we are unaware of the parent block is received (see New Blocks for more detail on receiving blocks)

Note that synchronization is initiated through `SyncService.startSync` which essentially resets the synchronization state and starts synchronizing the next batch of blocks through `SyncService.syncNext` that will generate a chain summary of relevant block IDs. These are then sent as a `BlockInventory` message of type `SYNC` (aka `SyncBlockChainMessage` message).

The receiving peer of such a message is handling the messages in `SyncBlockChainMsgHandler.processMessage`. Namely, it will find the last solid block that both know in the main chain and provide a list with the shared known solid block plus up to 2000 block IDs (up to the head) by sending a `ChainInventory` message (aka *ChainInventoryMessage*) which also includes the remaining number of unsent block IDs.

2.2.3.2.2 Asking For Blocks

As a result of receiving such a message, the `ChainInventoryMsgHandler.processMessage` will be eventually invoked. It will ensure that the block IDs received are ordered and have at least one element (would mean that the peers are fully synced) but at most 2001 block IDs (see above), however, requiring that if there are unsent IDs that the 2001 blocks have been fully filled. Further, they must overlap with the `SyncBlockChainMessage` sent before. All yet unknown blocks are then queued for fetching (`syncBlockToFetch`). Note that the peer during this execution time is marked as unfetchable. In case all block IDs have been shared, or if the configured block fetch limit has been reached, the generic fetch flag for the synchronization service's fetching thread is set to true. Alternatively, `syncNext` is repeated.

Hence, the `SyncService` has a thread for fetching the blocks. Namely, if the fetch flag has been set to true, then all idle and fetchable peers from which the node is syncing are asked for the blocks that they claimed they know. However, note that for each block ID only one peer is asked, limiting the number of asked for blocks to 100. Consequently, the threads send a `FetchInvDataMessage` message of type `BLOCK` to each of the peers.

2.2.3.2.3 Sending and Receiving Blocks For Synchronization

When receiving such a message a peer's `FetchInvDataMsgHandler.processMessage` will run which will ensure each block asked for is in a reasonable range. Ultimately, the block will be delivered (as long as it is available) and a `BlockMessage` is sent.

Consequently, the recipient of such a message will handle such a message in `BlockMsgHandler.processMessage` (note that at this point we elaborate only on the synchronization parts, see New Blocks for more). Only blocks that have been requested for synchronization will be accepted satisfying other validity properties. Ultimately, the requested block for synchronization is turned into a synchronized block to be processed. As such, the `SyncService`'s block processing flag is set to true, allowing the next scheduled block processing thread to run (a second type of thread owned by that service). Note that again, the synchronization will `syncNext` (ask for more fetchable items) or continue fetching more blocks.

2.2.3.2.4 Handling Synchronized Blocks

The aforementioned block processing thread in `SyncService` will be scheduled periodically and only perform the tasks if and only if the flag has been set (note it resets it immediately). The thread will iterate over all blocks queued for processing. Note that blocks with numbers lower or equal to the node's solid block, or blocks from disconnected nodes (e.g. due to an error) will be discarded. The witness signature is validated in the `TronNetDelegate.validateSignature` function and the block is processed through the `TronNetDelegate.processBlock` function that essentially pushes the block to the manager through `Manager.pushBlock`. Note that in case of errors, the node will disconnect from its peer.

2.2.3.2.5 Peers To Sync From

Note that additionally there is `EffectiveCheckService` which runs a thread that performs actions if all peers are syncing from the node. Namely, it chooses from the connectable nodes table offered by the p2plib a suitable node to connect to, to ensure that there is at least one peer from which it could sync if needed.

2.2.3.3 Transactions and Blocks

The publishing of transactions works by first advertising the hashes of transactions and blocks to peers who in turn respond by asking for a subset of the hashes advertised. Then, the advertiser shares the data. Below are more details.

2.2.3.3.1 New Transactions

Nodes can receive new transactions through the services they offer to users. For example, the `FullNode` runs the `RpcApiService` which is invoked by the auto-generated `WalletGrpc` upon reception of transactions and forwards the protobuf `Transaction` message to the `Wallet`. Similarly, other user-facing services will direct transactions to the `Wallet.broadcastTransaction` which will forward the transaction to:

1. the database manager `Manager.pushTransaction` that validates the signature, processes the transaction and pushes the transaction to the pending transactions (used in block production for example)
2. `AdvService.fastBroadcastTransaction` to send an `InventoryMessage` of type `TRX` containing the transaction to each connected non-syncing peer.

Note that transactions may also be received from other peers due to the advertising protocol (see Advertising). Namely, the `TransactionMsgHandler.processMessage` will accept asked-for transactions and queue them for multi-threaded processing. Note that smart contract interactions (creation and calls) are treated differently. Namely, they are queued for later processing by a periodically scheduled thread that will submit them to the queue mentioned before. Further, note that smart contract transactions may be ignored in case the queue has a size of 50.000. Thus, these transactions are considered to be a lower priority to the system. Similar to the user-facing publishing of transactions, the `TransactionMsgHandler` asks the `TronNetDelegate` to push the transaction to the database with `Manager.pushTransaction`. However, note that the transaction is not immediately broadcasted but queued for later broadcasting with `AdvService.broadcast`.

2.2.3.3.2 New Blocks

Note that a node may eventually produce a block. When such a block is produced, the block is queued for advertising similarly to a transaction in `AdvService.broadcast`. However, the transactions in the block are subsequently not advertised individually but rather as part of the block.

Additionally, other nodes will eventually communicate their blocks through the advertisement protocol (see Advertising).

Further, note that the block immediately triggers the processing of the queue. Similar to blocks received for synchronization, the requested blocks will be handled in `BlockMsgHandler.processMessage`. Equivalently, a maximum block size is enforced along with proper timing. Similarly, the signature of the block is checked and the block is queued for broadcasting and processed (recall `TronNetDelegate.processBlock` in Sending and Receiving Blocks For Synchronization). Note that the queue is then immediately worked on (the queue is shared with the transactions and hence, transactions will be worked on too).

Note that `startSync` may be started for the peer sending if the parent is unknown (see Synchronization).

2.2.3.3.3 Advertising

As seen in New Transactions, an *InventoryMessage* notifies peers about our new inventory. Transactions and blocks received on the P2P layer are also published in that fashion. More precisely, a node collects a batch of transactions and blocks to spread and advertise them to peers through `InventoryMessage` (one for transactions and one for blocks).

Eventually, a node will receive such `InventoryMessage` messages and handle them in the `InventoryMessageHandler.processMessage`. Blocks will always be queued for fetching while transactions could be dropped if there are more than 50.000 pending ones. Similar to sending advertisements, the response is not immediately sent for transactions but a separate periodically scheduled thread will send out `FetchInvDataMessage` messages. However, if a block inventory is received, sending out will be immediately started (again the same queue is worked on for both transactions and blocks, however, separate messages are sent).

Ultimately, as in the synchronization, `FetchInvDataMsgHandler.processMessage` will be responsible for receiving requests for publishing the inventory. Transactions will be collected and aggregated into one `TransactionsMessage` while blocks will be sent out individually as `BlockMessage`. Note that per hash only one peer is queried.

Note that additionally there is `FetchBlockService` which is a service that runs a periodically scheduled thread that tries to fetch the next block again if it has not been delivered fast enough.

2.2.3.3.4 Further Services

Note that the P2P layer consists of a few other services and functionalities.

1. `KeepAliveService`: Replies to `Ping` messages with `Pong` messages. However, the client never sends a `Ping`.
2. `NodePersistService`: A service that runs a thread that periodically stores the table of nodes to storage.
3. Statistics: Statistics for messages, peers and node statistics
4. `RelayService`: Essentially a service for witnesses that periodically tries to connect to relay nodes. Further, this includes special logic for relay nodes broadcasting (immediate broadcast) for blocks when they are received.
5. PBFT messaging: However, as of the trust model, it is expected to be not activated.

2.2.4 Assumptions

We make the following assumptions about the system:

- The number of 27 Super Representatives will not be changed. Theoretically, this number could be adjusted through a proposal.

- There is a parameter `vm.maxTimeRatio`, which is 5 by default. Each node can set its value for `vm.maxTimeRatio`. If there exists any transaction where the ratio between the slowest execution time on a witness and the fastest transaction time on a witness is bigger than `vm.maxTimeRatio` then consensus can fail. Hence, we assume that no such transactions exist.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0

High-Severity Findings	0

Medium-Severity Findings	3

- Denial of Service of Contract Creation Risk Accepted
- Extra Block Can Be Proposed During Maintenance Period Risk Accepted
- Timeout Checks Performed Synchronously Risk Accepted

Low-Severity Findings	4

- Ambiguous Ordering When Counting Votes of Witnesses Risk Accepted
- Events From Failed Reorganisation Not Removed Risk Accepted
- Known Inventory Will Be Fetched Risk Accepted
- intValue and longValue Do Not Behave as Documented Risk Accepted

5.1 Denial of Service of Contract Creation

Design Medium Version 1 Risk Accepted

CS-JTRON-009

When a contract is created by an externally owned account via the contract creation transaction, it is checked that there is no existing account at the address of the to-be-deployed contract. This check is performed by the function `create` of the `VMActuator`. If the check fails, an exception is thrown.

However, it is also possible to activate an account by sending some TRX or TRC-10 to it for example. A malicious actor could monitor the mempool for contract creation transactions and then front-run such transactions with a transfer to the address of the to-be-deployed contract. This would cause the check to fail and the contract creation to be denied.

Note that this attack is only possible if the contract creation transaction is sent by an externally owned account. If the contract is created by another contract with `CREATE` or `CREATE2`, the check is different and allows for the creation of a contract at an existing account as long as this account is not itself a contract.

Risk accepted

Tron answered:

```
TRON's packaging mechanism does not include a feature that allows transactions to be prioritized
by offering tips. Therefore, the success rate of such an attack by an attacker is relatively low,
and users can re-initiate transactions to deploy contracts after an attack. The impact is minimal,
so no modifications are made for the time being.
```

5.2 Extra Block Can Be Proposed During Maintenance Period

Correctness **Medium** **Version 1** **Risk Accepted**

CS-JTRON-010

The block validity check is not correct. As a result, the Super Representative in charge of performing the maintenance period can propose extra blocks during the extra 6 seconds of maintenance.

A misbehaving Super Representative can produce an extra block on top of the maintenance block, during the 6 maintenance seconds, which will be considered valid by the receiving nodes. `DposService.validBlock()` is responsible for asserting that the received block comes from the scheduled witness. It asserts that the absolute slot number of the received block is higher than the head block:

```
long timeStamp = blockCapsule.getTimeStamp();
long bSlot = dposSlot.getAbSlot(timeStamp);
long hSlot = dposSlot.getAbSlot(consensusDelegate.getLatestBlockHeaderTimestamp());
if (bSlot <= hSlot) {
  logger.warn("ValidBlock failed: bSlot: {} <= hSlot: {}", bSlot, hSlot);
  return false;
}
```

In the case of a block proposed during the extra 6 seconds of maintenance period, this assertion will pass, as `getAbSlot()` simply gets the number by dividing the time since genesis with the block generation period of 3 seconds.

Next `DposService.validBlock()` asserts that the scheduled block proposer of the current timestamp is indeed the signer of the block:

```
ByteString witnessAddress = blockCapsule.getWitnessAddress();
...
long slot = dposSlot.getSlot(timeStamp);
final ByteString scheduledWitness = dposSlot.getScheduledWitness(slot);
if (!scheduledWitness.equals(witnessAddress)) {
  logger.warn("ValidBlock failed: sWitness: {}, bWitness: {}, bTimeStamp: {}, slot: {}",
      ByteArray.toHexString(scheduledWitness.toByteArray()),
      ByteArray.toHexString(witnessAddress.toByteArray()), new DateTime(timeStamp), slot);
  return false;
}
```

Again the assert passes. This is because in `dposSlot.getSlot(maintenanceTime + 3000)` the `firstSlotTime` will be in the future, because `getTime(1)` accounts for maintenance periods.

```
public long getSlot(long time) {
  long firstSlotTime = getTime(1);
  if (time < firstSlotTime) {
    return 0;
  }
  return (time - firstSlotTime) / BLOCK_PRODUCED_INTERVAL + 1;
}
```

`getSlot(maintenanceTime + 3000)` returns 0. When querying `dposSlot.getScheduledWitness(0)`, the same witness as the previous block will be returned (if the active witnesses have not changed in the maintenance period).

The more general underlying issue is that `getScheduledWitness` returns incorrect information. The `getScheduledWitness` function of `DposSlot` is supposed to determine the scheduled witness for a particular slot that is passed to the function. However, this function does not check if the is a maintenance period between the last slot and the provided slot. Hence, it does not take into account that certain slots should be skipped. As a consequence, it can return incorrect results.

Risk accepted:

Client accepts the risk with the following statement:

```
At the end of each maintenance period, at most one SR can meet the conditions for such a
malicious action, and the maximum number of blocks that can be maliciously produced is two. The
produced blocks will be executed according to the normal logic and will not affect the
subsequent block production or the consistency of the network data. The impact is minimal, so no
changes are made for the time being.
```

We agree with Tron analysis of the issue. As a remark: In case there is a different attack that requires a Super Representative to propose multiple blocks in a row (which is otherwise hard), this finding provides a possibility to propose those consecutive blocks.

5.3 Timeout Checks Performed Synchronously

Design Medium Version 1 Risk Accepted

CS-JTRON-019

The TVM uses different timeouts. The timeout for block processing and the timeout for executing individual TVM transactions are checked synchronously. Hence, the actual execution times can significantly exceed the respective timeouts. This can happen if a single transaction or a single opcode take more time than expected. If an attacker manages to craft a transaction where the execution of a single opcode takes 10 seconds, then the honest block producer will patiently wait for this opcode to finish while generating the block. As a result, the block producer is unable to propose the block in a timely manner. They will only realize this after the long opcode has been executed.

Risk accepted:

Tron accepts the risk with the following statement:

```
At present, there is no situation where the execution time of a single instruction is too long, causing
the SR to produce blocks that time out and result in block loss. This issue is not actually present;
changing the timeout check to parallel execution would introduce unnecessary complexity and affect
node performance, so no changes will be made for the time being.
```

5.4 Ambiguous Ordering When Counting Votes of Witnesses

Design Low Version 1 Risk Accepted

CS-JTRON-024

During the maintenance period, votes for the witnesses are aggregated. After performing this vote aggregation and in order to extract the list of the 27 Super Representatives and 127 Standby Witnesses, the ordering is done according to number of votes first, and then by the `ByteString.hashCode()`. `hashCode()` has 4 bytes of entropy, and is not designed to be cryptographically secure. Hence, a witness could easily choose an address with a beneficial hashcode. Furthermore, a malicious witness could mine an address that has the same hashcode as another witness. Then, ordering could therefore be ambiguous.

Risk accepted:

Tron states:

```
The sorting rules are publicly transparent, and a stable sorting algorithm, TimSort, is used. When the
input source is stable and consistent, the output is also stable and consistent. The input source is
based on the witness vote counts and the hash values of the addresses, and the order is stable and consistent.
Based on this analysis, there is currently no unstable situation.
```

We remark that the order of the input of the sorting algorithm relies on the iteration order over the database values. For the supported databases, LevelDB and RocksDB, the values are iterated in lexicographical order of key, however, if another database engine were to be used, special care should be taken to ensure that database entries are iterated in the same order.

5.5 Events From Failed Reorganisation Not Removed

Correctness **Low** **Version 1** **Risk Accepted**

CS-JTRON-025

In `Manager.switchFork()`, before the blocks are being reverted, their "events" are removed:

```
reOrgContractTrigger();
reOrgLogsFilter();
eraseBlock();
```

However, if a reorganisation fails because a block in the new chain does not validate, then the new chain is reverted, but its events are not removed:

```
while (!getDynamicPropertiesStore()
    .getLatestBlockHeaderHash()
    .equals(binaryTree.getValue().peekLast().getParentHash())) {
  eraseBlock();
}
```

In this case, `reOrgContractTrigger()` and `reOrgLogsFilter()` are not called.

Risk accepted:

Tron states in response:

```
Event service developers should select the ultimately valid events based on BlockNumber and
BlockId, as well as the principle of the longest chain. This issue does not prevent event
service developers from correctly processing events. No action will be taken at this time,
and the documentation will be updated in the future.
```

5.6 Known Inventory Will Be Fetched

Design Low Version 1 Risk Accepted

CS-JTRON-028

Advertisements of blocks and transactions are fetched even if they are known.

Consider the following example:

1. Alice sends an inventory message for an old block to Bob.
2. Bob processes the message and queues it for fetching. Since it is a block inventory, Bob immediately tries to fetch it.
3. Alice delivers the block.
4. Bob will not do much as the received block will be below his head block number.

The reason for sending the message in step 2. is that the node checks only against the caches. Ultimately, unnecessary traffic is created.

Risk accepted:

Tron acknowledged the issue and accepted the risk. The following response was provided:

```
At present, duplicate checks have already been made for the latest xx blocks and transactions.
Attackers can only construct historical transactions and blocks for attacks. If it is a
historical block or transaction, the attacked node can simply process and discard it upon
receipt, without actually executing and consuming CPU resources. At the same time, the attack
will equally consume the attacker's bandwidth, so the overall impact is minimal. If existence
checks are added at this time, the opponent can construct some non-existent keys, which would
further affect system performance. In summary, no changes will be made for the time being.
```

5.7 `intValue` and `longValue` Do Not Behave as Documented

Correctness Low Version 1 Risk Accepted

CS-JTRON-022

In `DataWord`, the functions `intValue` and `longValue` are documented as throwing an `AritmeticException` if the values are too large to fit in the requested type. However, the implementation does not throw an exception, but instead returns the value modulo the maximum value of the requested type.

Because of this behavior, in the functions `freezeExpireTime`, `parseRessourceCode` and `convertResourceToString` of `Program`, if a number large enough is given as `resourceType`, it might be considered valid if, when truncated, it results in a valid resource type. The resource type obtained when truncating the number will then be used in those cases.

Note that if the functions were to throw when a given value is too large, the functions `intValueSafe` and `longValueSafe` would also throw since they respectively call `intValue` and `longValue` before making the overflow check.

Risk Accepted

Tron answered:

```
The error is only in the comments; the usage does not actually rely on the
``ArithmeticException`` being thrown as described in the comments. ``FreezeExpireTime``
and ``parseRessourceCode`` are only used after the Stake 1.0 proposal is activated, but
the Stake 1.0 proposal on the mainnet has been deprecated, so the issue described in the
report will not occur; the ``convertResourceToString`` method is used only for logging,
and its actual parameter usage is ``resourceType.sValue().byteValueExact()``, which will
throw an exception when the parameter is large.
```

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0

High-Severity Findings	4

- PBFT Messages Create State Expansion Code Corrected
- Resource Consumption by Blocks Not Signed by Witnesses Code Corrected
- Unbounded Memory Expansion in VOTEWITNESS Opcode Code Corrected
- Unpermissioned Censoring of Fork Blocks Code Corrected

Medium-Severity Findings	5

- Accounts Created With Suicide Are Not Charged Code Corrected
- Block Interval Not Enforced Specification Changed
- Forceful Disconnect via Relay Code Corrected
- Incorrect Address Comparison When Suiciding Code Corrected
- witnessStandbyCache Is Not Invalidated After Chain Reorganization Code Corrected

Low-Severity Findings	5

- Improper Cache Invalidation Code Corrected
- No Removal of Transactions to Spread if in Block Code Corrected
- Race Condition on Sync Block To Fetch Code Corrected
- Race-condition on Fetch Block Code Corrected
- Synchronization Issue During Block Generation Code Corrected

Informational Findings	1

- Redundant Map Clearing Code Corrected

6.1 PBFT Messages Create State Expansion

Security High Version 1 Code Corrected

CS-JTRON-004

While PBFT is expected to not be enabled, messages will nevertheless be accepted since an `allowPbft` check is not present.

If a `PbftCommitMessage` is received, the `PbftDataSyncHandler` will handle the message. Note that this will simply put the message into `pbftCommitMessageCache`. As a consequence, the size of the mapping will increase. The only place that touches the mapping otherwise, namely to remove items, is function `processPBFTCommitData` in the same class which is invoked during `SyncService.processSyncBlock`. However, it returns early when `allowPBFT` is `false`. Ultimately, the `pbftCommitMessageCache` will only grow. While normal nodes will not send such messages due to `allowPBFT` checks, malicious nodes may spam the memory of the honest nodes. Given the arbitrary-sized nature of the message, this may happen rather quickly.

In summary, an unprivileged peer can fill up the memory of any other peer. Over time this will lead to an `OutOfMemoryError` in the attacked node and can therefore crash the node.

Similarly, the `PbftMsgHandler` will accept PBFT messages, store information in memory, perform the requested operations and potentially forward PBFT messages to other peers. Hence, this could be an even more effective Denial-of-Service attack vector as the attacker does not need to be directly connected to the victim.

Code corrected:

The code has been corrected. More specifically, it is now enforced that the handlers mentioned above only process anything if PBFT is activated (which, as mentioned above, is not expected).

6.2 Resource Consumption by Blocks Not Signed by Witnesses

Security High Version 1 Code Corrected

CS-JTRON-007

When receiving a block, it is also processed if it is not signed by a witness. This can lead to significant resource consumption in two ways:

1. When receiving a block, it is checked in `BlockMsgHandler.processBlock()#141` whether the signer belongs to the witness set. If it is not in the witness set, the block broadcasting is delayed to after block processing in `tronNetDelegate.processBlock()`. The reason for doing so is that a valid block signer could not be in the witness set because of changes in the witness set happening in an ongoing maintenance period.

   If the block has an elapsed block number, it is considered a potential block belonging to a fork and is simply stored in the Khaos database. The validity of its signer within the witness set is never checked. The block is moreover broadcast to all the peers.

   Hence, overall anyone can generate a block that consumes plenty of resources, namely memory, storage, computing and network.

2. In the first phase of block processing, blocks are pushed (added to the database and considered for processing) as long as they have a valid signature. In the first phase of validation, it is not enforced that the signer is in the witness set (`TronNetDelegate.validBlock()` checks this, but just does not broadcast the block if the check fails). Whether the block is signed by the right witness is checked after all transactions of the block have been applied.

   When a longer chain is detected in `Manager.pushBlock()`, the current chain is reverted up to the level of the fork, and the blocks of the fork chain try to be applied (these blocks have not been validated yet). If applying the blocks from the fork chain fails, the original chain will be kept, and the fork chain discarded. This operation is however computationally very expensive because when `switchFork()` fails, all the original blocks have to be re-applied one by one. Note that this also prevents the affected node from generating blocks in this time.

   Since the blocks of the fork chain are not fully validated before `switchFork()` happens, the blocks do not need to be produced by an actual witness.

Code corrected:

The issue is addressed by filtering incoming blocks and dropping those that do not pass the signature check or whose signer is not in the active witnesses set. If these two conditions fail, the blocks are not processed.

The fix however potentially introduces race conditions where valid blocks are dropped because the signer is not yet in the client's view of the active witnesses, since the previous blocks updating the witnesses list might still be processing when the new one is received. Similarly, a block with a valid signature might be dropped, if the permissions of the signer have been changed in a previous block that still has not been processed.

Regarding this potential race condition, Tron comments:

```
The probability of this race occurring is quite low, and even if it occurs, the discarded
valid blocks can be retrieved again through block synchronization, without affecting block
consensus and nodes catching up blocks.
```

6.3 Unbounded Memory Expansion in VOTEWITNESS Opcode

Security **High** **Version 1** **Code Corrected**

CS-JTRON-005

When computing the cost of operation `VOTEWITNESS`, the memory expansion cost is underestimated by one. Passing a zero-length array at a high offset causes an unbounded memory expansion at a zero cost.

`VOTEWITNESS` takes 4 arguments:

- the offset in memory of the witnesses array
- the size of the witnesses array
- the offset in memory of the vote amounts array
- the size of the vote amounts array

The memory expansion costs of accessing those arrays are computed in `EnergyCost.getVoteWitnessCost()`:

```
BigInteger amountArrayMemoryNeeded = memNeeded(amountArrayOffset, amountArrayLength);
...
BigInteger witnessArrayMemoryNeeded = memNeeded(witnessArrayOffset, witnessArrayLength);
```

where `memNeeded()` returns `0` if `amountArrayLength` or `witnessArrayLength` are zero

If we pass to `VOTEWITNESS` zero-length arrays, it will therefore cost only the base Energy fee of 30000.

However, when executing the opcode, the following memory load is performed to validate the length of the array:

```
if (memoryLoad(witnessArrayOffset).intValueSafe() != witnessArrayLength ||
    memoryLoad(amountArrayOffset).intValueSafe() != amountArrayLength) {
    ...
```

Even if the length we passed as argument to the opcode is zero, the array offset is still accessed, because the operation assumes that the first memory word of the array is the size and wants to validate it. This access is not priced in `getVoteWitnessCost()`. If `witnessArrayOffset` or `amountArrayLength` are very high, this will trigger a huge memory expansion, which costs 0 energy to the user.

This issue may incur an attack with a transaction of more than 200ms execution time. Additionally, the resulting garbage collection can consume a significant amount of time. While the issue could cause a Denial-Of-Service, it is rather unlikely due to the precise timings required. Such a Denial-Of-Service could occur as described below.

The witness which receives the transaction wants to broadcast it. As part of that broadcast, it calls `pushTransaction` in `Manager.java#831`. This takes some locks and then calls `processTransaction`. For this transaction, `processTransaction` could take for example 2.5 seconds as it is being delayed by garbage collection. During these 2.5 seconds, the witness would like to start generating a new block, but it can't because it is waiting for the lock.

Eventually, `processTransaction` finishes and the transaction is added to `pendingTransactions` and the block generation can obtain the locks. The relevant part of the block generation is in `Manager.generateBlock`. Here, the witness:

- Takes out the attack transaction out of the `pendingTransactions` using `pendingTransactions.poll();`
- Then the witness does the timeout check `if (System.currentTimeMillis() > timeout) {` and immediately breaks from the loop.

Hence, the witness removed the transaction from pending but did not put it into any block. This process continues with the next witness. The attack transaction is re-pushed but not included in a block. As previously mentioned, this is very unlikely to work due to the precise timings needed for receiving, processing and broadcasting the transaction as well as the required timing of the garbage collection.

Code corrected:

Version 2 introduces parameter #81 `ALLOW_ENERGY_ADJUSTMENT`, which updates the energy cost calculation of `VOTEWITNESS` to use `Energy.getVoteWitnessCost2()` instead of the flawed `EnergyCost.getVoteWitnessCost()`. `getVoteWitnessCost2()` correctly accounts for the length of the witnesses and votes arrays, which include an initial word for the size of the arrays. Proposal #91 aims to enable the `ALLOW_ENERGY_ADJUSTMENT` parameter.

6.4 Unpermissioned Censoring of Fork Blocks

Security High Version 1 Code Corrected

CS-JTRON-006

An attacker can censor fork blocks received by a node, by making the node "forget" those. As seen in the issue Resource Consumption by Blocks not signed by witnesses the `switchFork()` functionality can be triggered with "fake" blocks by an unprivileged attacker. In that case, `switchFork()` will fail, and the original chain will be kept, after considerable computational effort. What also happens when `switchFork()` fails is that the complete fork chain is removed from the Khaos database of blocks. This means that an attacker can censor a fork block received by a node by building a chain on top of it. To achieve this the attacker builds on the fork block with fake blocks, that will trigger a `switchFork()`. When the fork switching fails (because the blocks from the attacker are invalid), also the legitimate block in the fork gets removed from the node database.

`switchFork()` at line 1117-1121 in Manager.java:

```
if (exception != null) {
  ...
  logger.warn("Switch back because exception thrown while switching forks.", exception);
  first.forEach(khaosBlock -> khaosDb.removeBlk(khaosBlock.getBlk().getBlockId()));
```

Code corrected:

This issue is resolved by dropping blocks from invalid producers before they are processed. The fix for Resource Consumption by Blocks not signed by witnesses also addresses this.

6.5 Accounts Created With Suicide Are Not Charged

Design Medium Version 1 Code Corrected

CS-JTRON-002

According to the specifications of Tron, creating an account on the Tron network requires paying a fee. This provides important protections. However, if a contract executes the `SUICIDE` opcode and the inheritor address does not refer to an existing account, then an account is created without the need to pay a fee.

It could be possible for a malicious user to spam the suicide opcode in a transaction with different addresses as inheritors to make the state grow a lot. A similar attack was performed against the Ethereum network in 2016 where the attacker did the following:

Given a contract A with the following code:

```
PUSH1 0x00
CALLDATALOAD
SELFDESTRUCT
```

Then, deploy another contract `B` that, when called, calls `A` repeatedly, each time with a different calldata. In that setting the attacker still needs to pay the energy cost of contract creation, but when performing multiple `SUICIDE` operations, this account creation is still significantly cheaper than the regular account creation. Furthermore, if the attacker has unused energy from their stake, then they can create accounts at no additional costs.

Overall, the attacker would then just spam transactions to contract `B` to create a large amount of new accounts. That would significantly increase the blockchain's state size.

Code corrected

Version 2 introduces parameter #81 `ALLOW_ENERGY_ADJUSTMENT`. In the case, the corresponding Committee proposal #91 about allowing the adjustment on Energy consumption of TVM instructions is activated, the gas function used for the `SUICIDE` instruction is `getSuicideCost2`, which charges `NEW_ACCT_CALL` extra energy in the case the inheritor address does not refer to an existing account. This behavior is consistent with the rest of the TVM instructions, which charge extra energy when creating a new account such as `CALL`.

Furthermore, regarding the state increase, Tron comments:

```
Due to the different storage mechanisms of TRON and Ethereum, the increase in TRON's 'state
size' does not significantly degrade data performance. This attack method used on Ethereum does
not have a major impact on TRON.
```

6.6 Block Interval Not Enforced

Correctness Medium Version 1 Specification Changed

CS-JTRON-008

The documentation says:

```
In the TRON network, the block interval is 3 seconds, that is, a block is generated every 3 seconds.
```

The `validBlock` function of `DposService` is in charge of checking the correct timestamp. However, it uses `getAbSlot` which rounds down the time difference. Hence, a block produced by a single, malicious witness could lead to a block interval between one and five seconds. All other nodes would accept that malicious block, due to the check discussed above. Hence, blocks are not guaranteed to appear every three seconds if there is a malicious witness.

Specification changed:

Tron updates the specification of the system to include the reported behavior and states:

```
In the first instance where an SR does not engage in malicious behavior, the block
production logic ensures that the block time is a multiple of three. If an SR were
to maliciously alter the block header time, they could only modify it to be within
0 to 3 seconds of the current time. Additionally, if the block header time is not
a multiple of three, it will not affect the data consistency of the chain and will
not interfere with the normal block production by the next SR. Furthermore, since
EVM chains like Ethereum do not have a fixed block time, smart contract transactions
generally do not rely on a fixed periodic block time. Considering the overall impact
is minimal, we will not make changes at this time, but we will update the
documentation in the future.
```

We remark that the `block.timestamp` observed from the TVM can deviate up to 2 seconds from the expected one. No smart contract application should rely on the timestamp spacing being constant.

6.7 Forceful Disconnect via Relay

Security Medium Version 1 Code Corrected

CS-JTRON-003

The `BlockMsgHandler.processMessage` function performs `check` as follows:

```
if (!fastForward && !peer.isRelayPeer()) {
  check(peer, blockMessage);
}
```

Meaning, if a relay node has sent a block message or if the node itself is a relay node, the check will not be performed.

The `check` may however trigger a disconnect. For example, as follows:

```
if (blockCapsule.getInstance().getSerializedSize() > maxBlockSize) {
  logger.error("Receive bad block {} from peer {}, block size over limit",
          msg.getBlockId(), peer.getInetSocketAddress());
  throw new P2pException(TypeEnum.BAD_MESSAGE, "block size over limit");
}
```

Now the following scenario could occur:

1. The attacker sends a block to the relay node and that block is too big.
2. The relay node does not `check` the block size.
3. The relay node eventually forwards it to its peers.
4. The relay peers will have the relay stored and also skip the `check`.
5. The relay peers will forward the block to their peers.
6. These peers will disconnect due to a too-big block message.

In conclusion, the attacker can send specially crafted blocks to relay nodes. The relay nodes and the relay peers will accept them. However, once the relay peers forward them to their peers they get disconnected from all their peers. Hence, a single block could cause a great wave of disconnects.

Code corrected:

The size check is now done for every peer. Similarly, is the `gap` check. Ultimately, `check` will only check peer-specific constraints which do not apply to relay nodes.

6.8 Incorrect Address Comparison When Suiciding

Correctness **Medium** **Version 1** **Code Corrected**

CS-JTRON-012

In the function `suicide` of `org.tron.core.vm.program.Program`, the following comparison is made:

```
FastByteComparisons.compareTo(owner, 0, 20, obtainer, 0, 20) == 0
```

It compares the first 20 bytes of the `owner` and `obtainer` arrays. However, both arrays are 21 bytes long. This means that the last byte of the `owner` array is never compared. If a contract suicide with an inheritor whose address's first 20 bytes match the first 20 bytes of the contract, the balance of the contract will be sent to the black hole and lost. As a match of 20 bytes is required, the probability is extremely small.

Code corrected

In the case the Committee proposal #91 about allowing the adjustment on Energy consumption of TVM instructions is activated, the comparison is done on 21 bytes and not 20 bytes.

6.9 witnessStandbyCache Is Not Invalidated After Chain Reorganization

Correctness **Medium** **Version 1** **Code Corrected**

CS-JTRON-023

The `WitnessStore` implements a custom cache logic, by instantiating the `TronCache` object `witnessStandbyCache`. When `getWitnessStandby()` is queried, in the block reward distribution logic of `MortgageService`, the cache is queried. However, the cache is not invalidated when a chain reorganization happens and a database snapshot is retreated. This means that after a reorganization, which involves a maintenance period that changed the standby witness list, the standby witness cache will not be rolled back, and the block rewards will be distributed incorrectly. This can break the consensus, as the nodes who reorganized the chain will attribute the rewards incorrectly, and the nodes who did not reorganize will attribute them correctly. The difference in state will only be obvious when a transaction involving amounts from incorrectly attributed rewards will be rejected by some nodes but not others.

Code corrected:

The `witnessStandbyCache` has been deprecated. Instead, the standby witnesses list is computed on the fly at the end of the processing of every block, when the payouts to witnesses are distributed.

6.10 Improper Cache Invalidation

Design | Low | Version 1 | Code Corrected

CS-JTRON-026

Caches in `PeerConnection` are cleaned up with `cleanUp` as part of the `onDisconnect` function. Using `cleanUp` means that only expired items are evicted. However, the `invalidateAll` function could be used as it performs an actual cleanup. The function `invalidateAll` is also used in other places e.g., `TronCache`.

Code corrected:

The function `invalidateAll` is now used.

6.11 No Removal of Transactions to Spread if in Block

Design | Low | Version 1 | Code Corrected

CS-JTRON-030

When a `BlockMessage` is broadcasted in `AdvService.broadcast`, the function tries to remove the transactions in the block from `invToSpread` as follows:

```
Sha256Hash tid = transactionCapsule.getTransactionId();
invToSpread.remove(tid);
```

However, note that `invToSpread` has type `Item` as key which will essentially have the same `hashCode` function return value. However, due to the `ConcurrentHashMap` supporting multiple values per bucket in the hash map, the removal is also based on equality. Namely, a key is only removed if `keyInMap.equals(toRemoveKey)`.

However, `Item` defines the `equals` function to be `false` if the runtime type of the compared to object is not the same. Hence, the transaction is not removed from `invToSpread`, leading to publishing transactions that had been published in a block.

However, that does not impact communication with other peers significantly.

Code corrected:

The operation, which tried to remove the transaction from *invToSpread* was deleted. Please note that Tron has clarified that not removing the transaction is acceptable behavior because it does not create a significant network overhead.

6.12 Race Condition on Sync Block To Fetch

Design | Low | Version 1 | Code Corrected

CS-JTRON-033

When for example `PeerCheckStatus` detects that a node has been disconnected, it will call `PeerConnection.onDisconnect` which will `clear` the `PeerConnection.syncBlockToFetch` `Deque`. However, other processes may be running concurrently that could be affected, some of which may result in unhandled exceptions. Namely, the thread in `SyncService` running `handleSyncBlock` will pop items from the `Deque`. Hence, if the executions overlap so that the `Deque` is cleared right before the `pop`, the `pop` will raise `NoSuchElementException` due to it being empty. Ultimately, an unexpected exception may occur.

Code corrected:

The code has been corrected. More specifically, `pop` is no longer used. Instead `peek` and `remove` are used. Thus, the program will ensure now that the ID is removed but will tolerate removals by other threads.

6.13 Race-condition on Fetch Block

Design | Low | Version 1 | Code Corrected

CS-JTRON-034

In `FetchBlockService`, the property `fetchBlockInfo` may be changed by several threads which creates race-conditions on the variable due to a lack of synchronization.

For example, the function `fetchBlock` is called by an `InvSender` (ultimately by the fetching process in `AdvService`) to store the fetch block if needed and possible. Now, consider the following example.

1. Thread A handles a `InventoryMessage` from P1 for a block corresponding to head + 1.
2. Thread B handles a `InventoryMessage` from P2 for a block corresponding to head + 1.
3. Thread A processes the inventory queue first and sends the fetch request for the block.
4. Thread B processes the inventory queue after and sends the fetch request for the block.
5. Thread A's `fetchBlock` sees the `fetchBlockInfo` to be `null` and sets it.
6. Thread A's modification to memory has not become visible yet to Thread B's. Thus, the latter set `fetchBlock` sees the `fetchBlockInfo` to be `null` and sets it.

Ultimately, the value has been overridden.

Similarly, the following may occur.

1. Assume a the block fetching process `fetchBlockProcess` is running and wants to set the fetch block info to null.
2. However, before the setting to null occurs, `blockFetchSuccess` sets it to null due to a received message.

3. The fetching thread `consumerInvToFetch` calls `fetchBlock` with the current head + 1. It sees the fetch block info being null and proceeds to set it to head + 1.
4. Now, however, the thread from 1. sets it back to null.
5. The thread from 3 now tries to log. However, if the memory has been pushed by the other thread, `fetchBlockInfo.getPeer()` in `fetchBlock` will cause a null pointer exception.

While this scenario is rather unlikely due to the particular timing requirements and the requirements that the threads will push their local memory nearly immediately to the shared memory, it nevertheless illustrates the race-condition. Note that this particular scenario could throw an exception, leading to a loss of knowledge about advertised blocks and transactions stored in `AdvService.invToFetch`.

Ultimately, there is a race-condition on `fetchBlockInfo`.

Code corrected:

The code was updated to cache some of the data within the scope of the corresponding threads. With that change, the race condition is technically still present, but it only has a minimal (GC-related) effect now. The risk for a null pointer exception is gone.

6.14 Synchronization Issue During Block Generation

Design | Low | Version 1 | Code Corrected

CS-JTRON-018

When a node starts generating a new block, it executes the following code inside the function `DPosTask.produceBlock`:

```
State state = stateManager.getState();
if (!State.OK.equals(state)) {
  return state;
}

...
try {
  synchronized (dposService.getBlockHandle().getLock()) {
```

Hence, it will first check the state and then start generating a block, if the state is `OK`. When a block verification of an incoming block is currently in progress, this function will have to wait at the `synchronized` line for the block verification to finish. However, that block verification might change the state to no longer be `OK`. But `produceBlock` does not check `getState` again. In this scenario, the node would start generating the block even though the state is not `OK`.

Code corrected:

When generating a block, the state check is now performed after acquiring the synchronization lock. A state check is also performed before acquiring the synchronization lock to avoid unnecessary lock contention.

6.15 Redundant Map Clearing

Informational Version 1 Code Corrected

CS-JTRON-038

In the function `PeerConnection.onDisconnect` the `clear` function of the variable `syncBlockInProcess`, which is a `HashSet`, is called twice back-to-back.

Code corrected:

The code has been adjusted to only `clear` once.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Incorrect Comment

Informational Version 1 Acknowledged

CS-JTRON-036

The variable `TronNetDelegate.hitDown` has a comment indicating that it is used for tests. However, the `exit` variable is used for tests. The comment appears to be incorrect.

Acknowledged:

Tron acknowledged the issue and stated:

```
The comment was indeed misplaced.
```

7.2 Redundant Channel Closing

Informational Version 1 Acknowledged

CS-JTRON-037

When the function `PeerConnection.disconnect` is called, it automatically closes the channel from the `p2plib` by calling `Channel.close`. However, in `PeerManager.close` the peer is disconnected but also the channel is closed which is redundant. Similarly, that is the case in `RelayService.disconnect`.

Acknowledged:

Tron replied:

```
It is indeed closed twice. The second close will not incur any error and occupies very few
resources, so it has minimal impact.
```

7.3 Redundant and Inconsistent Check

Informational Version 1 Acknowledged

CS-JTRON-039

When the `BlockMsgHandler` processes the block, it checks for its validity. However, when checking the block timestamp there are two redundant checks.

The first check validates that the block time is not 3 seconds or more in the future in `BlockMsgHandler.check`:

```
long gap = blockCapsule.getTimeStamp() - System.currentTimeMillis();
if (gap >= BLOCK_PRODUCED_INTERVAL) {
  logger.error("Receive bad block {} from peer {}, block time error",
          msg.getBlockId(), peer.getInetSocketAddress());
  throw new P2pException(TypeEnum.BAD_MESSAGE, "block time error");
}
```

However, the `processBlock` function will validate the block with `TronDelegate.validBlock` which will check the timestamp a second time. It will check that the block timestamp is not more than one second in the future:

```
if (block.getTimeStamp() - time > timeout) {
  throw new P2pException(TypeEnum.BAD_BLOCK,
          "time:" + time + ",block time:" + block.getTimeStamp());
}
```

where `timeout` is set to 1000 milliseconds.

Acknowledged:

Tron acknowledged the issue and stated:

```
The block time check is performed twice, which has minimal impact, and one of the checks can be removed
```

7.4 TRC-10 Information Is Not Supported by Delegatecall

Informational **Version 1** **Acknowledged**

CS-JTRON-016

When a contract does a `DELEGATECALL`, the current context's `callvalue` and `sender` are passed to the subcontext. This is not the case for the `calltokenvalue` and `calltokenid`.

Acknowledged

Tron answered:

```
TRC-10 is a feature on TRON that is not commonly used. The TRON Virtual Machine (TVM)
provides basic support for TRC-10, but the ``delegatecall`` operation does not support
TRC-10.
```

7.5 Undocumented and Unused Fields and Properties

Informational **Version 1** **Acknowledged**

CS-JTRON-040

The Tron protobuf protocol documentation is outdated as some fields remain undocumented. For example, `HelloMessage` has a `nodeType` and a `lowestBlockNum` which are undocumented.

Similarly, block inventory messages can be of type `ADVTISE` which is unused.

Additionally, many message types (see `MessageTypes`) are unused such as `BLOCKS`, `BLOCKHEADERS`, `ITEM_NOT_FOUND`, `FETCH_BLOCK_HEADERS`, `TRX_INVENTORY` and more.

Several parameters in `CommonParameter` are unused. For example, `nodeChannelReadTimeout`, `tcpNettyWorkThreadNum`, `udpNettyWorkThreadNum` or `receiveTcpMinDataLength`. Also, `p2pDisable` is always false.

The setter for `PeerConnection.isRelayPeer` is unused.

Similarly, there are unused functions such as `getHeadBlockTimeStamp` or `getGenesisBlock` in `TronNetDelegate`.

Additionally, the timestamp in `syncChainRequested` is unused.

Acknowledged:

Tron acknowledged the issue and stated:

```
TRON's protobuf document is outdated.  Since some of the fields are still not documented,
many fields in variables are not being used.
```

7.6 Wrong Reason Code

Informational **Version 1** **Acknowledged**

CS-JTRON-041

When processing a `HelloMessage` inside the function `processHelloMessage` of `HandshakeService` a check is made:

```
if (!msg.valid()) {
  ...
  peer.disconnect(ReasonCode.UNEXPECTED_IDENTITY);
```

The reason code `UNEXPECTED_IDENTITY` is returned in the error case but does not really make sense as nothing identity-related was checked in `valid`.

Acknowledged:

Tron acknowledged the issue and stated:

```
The disconnection process used an incorrect verification code; the correct verification code should
be used instead.
```

7.7 `getAnswerMessage` Is Not Set for Some Messages

Informational **Version 1** **Acknowledged**

While the function `getAnswerMessage` of classes of type `Message` is not used, it can help understand the message types. However, not every class has the expected return message returned by the method. For example, `FetchInvDataMessage` returns `null` but the expected answer message is either a list of `BlockMessage` or `TransactionsMessage`.

Code corrected:

Tron acknowledged the issue and stated:

```
GetAnswerMessage is not being utilized, it will be removed.
```

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Capabilities of Powerful Attacker

Note Version 1

Throughout our work, mostly a malicious witness was the strongest attacker we considered. However, an attacker that has stronger network capabilities than the average node (e.g., an ISP) may attack the network.

Consider the following examples:

1. If the attacker can repeat messages: The attacker could repeat hello messages to forcefully disconnect two nodes due to two hello messages being delivered. The sender of the duplicated hello would be banned for 1 hour.
2. If the attacker can interrupt messages: The attacker could censor the delivery of certain inventory so that the delivery times out and the peers disconnect. The sender of the inventory would be banned for 1 minute.
3. If the attacker can modify messages or forge them: The attacker could provide bad transactions to provoke a disconnect. The seemingly malfunctioning sender would be banned for 1 hour.

Overall, an attacker with such strong network capabilities could relatively easily perform eclipse attacks and other network-based attacks.

8.2 Parameter Range Dangerous for Consensus

Note Version 1

Node operators can choose the value of the `BLOCK_PRODUCE_TIMEOUT_PERCENT` parameter. This parameter indicates which percentage of the **half** block time (so currently 1.5 seconds) can be spent for transaction processing inside block generation. A node operator can set this to 100%, however, 100% never really makes sense as we will argue below.

For this example, we assume that there are many pending transactions. If the `BLOCK_PRODUCE_TIMEOUT_PERCENT` is set to 100%, the transaction processing for block generation will take more than 1.5 seconds (because it will only be stopped after it crosses the timeout). The total block generation time will be even larger as it also includes, computing the merkle tree and signing the block. Then the block needs to be broadcast. Even in the optimal case, where the next active witness is a direct peer of the current active witness, this broadcast step will take some time. Then the block verification at the new witness starts. Part of this verification is again the transaction processing of 1.5 seconds, but additional steps such as signature verification are also necessary.

Hence, the two steps of transaction processing already require more than the block time. Additionally, time is needed for signature generation, signature verification, merkle tree computation, merkle tree verification, broadcast and many more steps. In conclusion, even in the ideal case, significantly more than the block time is needed. Hence, the next active witness can only start generating its block with a delay, as it has to wait for the block verification to finish.

The situation is even more severe when the two witnesses are not peers. Then the broadcast time is a lot higher as the intermediate nodes perform checks before forwarding the block. In this scenario, it is plausible that the next witness will not receive the block in time and hence will not build on top of it.

8.3 Voting More Effectively Against Proposals by Late Approval Retraction

Note Version 1

Proposals are often accepted with about 18 to 22 votes in favor. Additional voters seem to encounter voter fatigue or do not want to pay for voting. Hence, voters against a proposal might be able to vote more effectively, by first voting in favor of a proposal, just to retract their vote in the last block of the cycle, before proposal processing. Then, the proposal might not pass the necessary quorum.