Security Audit Report

Celestia Q2 2025:
High Throughput Recovery

Last Revised
2025/06/24

Authors:
Martin Hutle, Sergio Mena, Tatjana Kirda, Marius Poke

Contents

Audit overview

The Project

In April and May 2025, Celestia engaged Informal Systems↗ to work on a partnership and conduct a security audit of the following items:

1. Repository `celestia-core`, branch `feature/recovery`
   - celestia-core/consensus/propagation↗
2. Repository `celestia-app`, branch `evan/spec-PBBT`
   - Vacuum! Part I: High Throughput Recovery specification

Relevant code commits

The audited code was from:

- Branch `feature/recovery`
  - commit hash `139bad235a379599670f30d5e28c637dde4bb17a`↗
- Vacuum! Part I: High Throughput Recovery specification
  - commit hash `a0369f750aed5129f37d93f0f65c234ee1f10d12`↗

Scope of this report

The primary focus of this audit was the High Throughput Recovery algorithm, which is built on top of CometBFT as a new reactor to enhance block propagation.

As part of this inspection, the data types and algorithm defined within the propagation reactor were examined in detail. The code review also covered the integration points of the propagation reactor with the consensus code.

The scope of this audit did not include a review of the existing CometBFT logic or the test code. Furthermore, it did not cover the mempool, end-to-end tests, or cryptographic components.

Audit plan

The audit was conducted between April 28th, 2025, and May 30th, 2025 by the following personnel:

- Martin Hutle
- Marius Poke
- Sergio Mena
- Tatjana Kirda

Conclusions

The audit was conducted while the code was still in development. Consequently, prior to the audit, the clients shared several known issues with the current implementation. The full mesh overlay has not yet been implemented, leaving the system vulnerable to determined attackers. Additionally, the compact block lacks signature verification, making it susceptible to forgery attacks. The calculation of the PerPeerConcurrentRequestLimit assumes that voting power is evenly distributed among validators. The specification also envisions an ideal protocol without per-peer bandwidth restrictions.

During the audit, we identified thirty-one findings, including three of critical severity, three of high severity, seven of medium severity, twelve of low severity, and six informational. Full details of these issues can be found on the Findings page.

Audit Dashboard

Target Summary

- **Type:** Protocol and Implementation
- **Platform:** Go
- **Artifacts:**
  - High Throughput Recovery algorithm implementation, branch `feature/recovery`↗
    → celestia-core/consensus/propagation↗
  - Vacuum! Part I: High Throughput Recovery specification↗

Engagement Summary

- **Dates**: 28.04.2025 - 30.05.2025.
- **Method**: Manual code review, protocol analysis

Severity Summary

Finding Severity	Number
Critical	3
High	3
Medium	7
Low	12
Informational	6
Total	31

System Overview

High-Level Algorithm Description

High Throughput Recovery activates after the initial dissemination of transactions. The proposer node broadcasts a *compact block* — a summary, or TOC, of the proposed block — to all other nodes in the network. This compact block enables other nodes to reconstruct the full block using various techniques, without requiring full transmission.

Block reconstruction relies on the block parts mechanism defined in CometBFT, and employs the following methods:

- **Local Mempool Reconstruction:**

  The propagation reactor queries the mempool reactor. For every transaction in the proposed block, the compact block includes the transaction's hash, and its start and end offsets in the serialised block. If enough transactions are found locally, nodes can reconstruct one or more block parts from them.

- **Parity-Based Recovery:**

  The proposer generates *parity parts* using the `reedsolomon` library. With a suitable combination of block parts and parity parts, missing parts can be reconstructed through erasure coding.

- **Pull-Based Retrieval:**

  A tree-based pull mechanism — the core of High Throughput Recovery — enables nodes to fetch missing parts from peers.

The algorithm also includes a **load balancing strategy** to avoid congestion at the proposer node, which acts as the root of the retrieval tree.

The data structures involved include mechanisms for verifying messages and block parts. These are in scope for the audit.

Catchup protocol

The catchup protocol is triggered for all heights `h` smaller than the current local height of the consensus protocol. If there is a block part `bp[h][r]` missing for the last round `r` of that height, request this part from a peer from which is has not been requested before (`WantParts`).

Once a block part is received, it is handled as in the normal case protocol, except that only normal parts (and not the parity parts) are requested and used to reconstruct the block.

The protocol is triggered at the following places:

- Every `RetryTime` ticks
- When consensus enters the precommit phase for height `h` and round `r`
- When consensus enters the commit phase for height `h` and round `r`
- When consensus adopts a new valid block for height `h` in round `r` (`2f+1` prevotes)

Assumptions and Scope

The algorithm assumes a **stable network topology** during the processing and dissemination of each proposed block. This means the node availability and communication links are fixed during that period. If dynamic changes occur mid-block (e.g., connection to a peer shuts down), the algorithm does not account for them explicitly. This modeling choice keeps the threat model more concise and focused, and reflects the absence of topology-adaptive logic in the provided specification.

A general understanding of **CometBFT** is assumed. Concepts such as validators, block proposals, consensus rounds, and message propagation are not reintroduced here. Readers unfamiliar with these should consult the CometBFT documentation.

Notation and Conventions

The threat model uses standard notation, formal logic terminology, and first order logic expressions. It is structured as a set of properties that correct nodes must follow at all times (also called invariants). A list of threats are derived from the properties, which guide our security analysis.

The following conventions apply:

- **"iff"** (if and only if) is used in two ways:
  - In definitions: "A iff B" means A and B can be used interchangeably.
  - In properties: "A iff B" means "A if B and B if A".
- **"Previously"** is a tricky concept in a distributed system, and is interpreted according to context:
  - For events on the same node, it refers to real-time order.
  - For events on different nodes, it implies a causal link via message passing.
- Indices in arrays or lists are **zero-based**.
- Familiarity with **first-order logic** and **partially synchronous distributed systems** is assumed.

Threat Model

Threat model for pull-based broadcast tree recovery

This report presents a threat model for **High Throughput Recovery**, an algorithm added to **CometBFT** as a new reactor to enhance block propagation. The threat model focuses on the behaviour of **correct nodes**; unless otherwise specified, the term *node* refers exclusively to a correct node. When we refer to a Byzantine or malicious node, it is stated explicitly.

Properties

This section outlines the threat model: a set of definitions, rules, and properties that specify how the system should behave on honest nodes. The threat model serves as a basis for deriving the list of threats, presented in section 'Threats', which was used to guide our audit of the code in our search for issues. The threat model and its associated threats enable a methodical and comprehensive analysis, as opposed to simply reviewing the code without a concrete roadmap.

This section is organised as follows. Firstly, we introduce several type definitions to enhance the readability of the remainder of the section. The remainder is then divided into two parts: in the first part, we explicitly state the properties that govern the data structures utilised by High Throughput Recovery; in the second part, we set out the properties that the algorithm itself must satisfy. Each part is preceded by a subsection containing definitions that contribute to a clearer organisation of the text.

Basic type definitions

In order to improve readability, we use the following terms in this document:

- *hash*, a fixed-size byte array in the code
- *height*, a blockchain height, `int64` in the code
- *round*, a Tendermint consensus round, `int32` in the code
- *offset*, an offset in a byte array, `uint32` in the code
- *length*, the length, or an index, of an array, `uint32` in the code
- *bitmask*, an array of bits
- *merkle proof*, a data structure containing the aunts that prove inclusion of an element into a tree
- *block id*, a structure containing
  - *total*, the number of block parts of a serialised block
  - *hash,* a hash containing the root hash of the parts

Data flow definitions

We define `ProposedBlock` as a structure composed of:

- `height`, a height
- `round`, a round
- `txs`, an array of arrays of bytes
- (all other fields are not relevant for this threat model)

We define `Proposal` — unchanged from the original CometBFT/Tendermint Core — as a structure composed of:

- `type` (not relevant for this threat model)
- `height`, a height
- `round`, a round
- `pol_round` (not relevant for this threat model)
- `block_id`, a block id
- `timestamp` (not relevant for this threat model)
- `signature`, an array of bytes

We define `TxMetaData` as a structure composed of:

- `hash`, a hash
- `start`, an offset
- `end`, an offset

We define `RecoveryPart` as a structure composed of:

- `height`, a height
- `round`, a round
- `index`, a `uint32`
- `data`, a byte array
- `proof`, a merkle proof

We define `CompactBlock` as a structure composed of:

- `bp_hash`, a hash
- `blobs`, an array of `TxMetaData`
- `proposal`, a `Proposal`
- `last_length`, a length
- `parts_hashes`, an array of hashes
- `signature`, a signature

Given a `CompactBlock` instance `cb`, we define the function *signBytes(cb)*, which returns a fixed-size byte array containing the cryptographic signature covering the following fields:

- `cb.bp_hash`, `cb.blobs`, `cb.last_length`, `cb.parts_hashes`
- notice that `cb.proposal` is excluded as it is itself signed

We define `PartMetaData` as a structure composed of:

- `index`, a length
- `hash`, a hash

We define `HaveParts` as a structure composed of:

- `height`, a height
- `round`, a round
- `parts`, an array of `PartMetaData`

We define `WantParts` as a structure composed of:

- `height`, a height
- `round`, a round
- `parts`, a bitmask

- `prove`, a boolean

Data Flow 1: `ProposedBlock` and `Proposal` as input

Inputs => Outputs:

- *Proposed block*, `Proposal` => `CompactBlock`
- *Proposed block*, `CompactBlock` => `RecoveryPart`

DF1a. A `ProposedBlock` (or *proposed block*) *b* can be serialised into an array of bytes, denoted as *serialised(b)*:

- The serialisation follows ProtoBuf rules.
- During the serialisation process, we record the start and end byte offsets for each element of *b.txs*. For any transaction *b.txs[i]*:
  - we denote *start(b, i)* as the offset of *b.txs[i]*'s first byte in *serialised(b)*
  - we denote *end(b, i)* as the offset of *b.txs[i]*'s last byte in *serialised(b)*

DF1b. Proposed block *b* can be used as input to create an array `tms` of `TxMetaData`:

- `tms[i].hash` is the hash of *b.txs[i]*
- `tms[i].start` is *start(b, i)*
- `tms[i].end` is *end(b, i)*

DF1c. A serialised proposed block *serialised(b)* can be divided into a list of *n* parts, denoted as *parts(serialised(b))*:

- Each part is a byte array
- `BlockPartSizeBytes` is a constant that defines the length in bytes of each part, except for the last part, which may be shorter

DF1d. The parts of a serialised proposed block, *parts(serialised(b))*, can be extended into redundant parts *parity(serialised(b))*:

- We pad the last part of *b* to size `BlockPartSizeBytes`
- We use the `reedsolomon` library to produce as many parts of *parity(serialised(b))* as *len(parts(serialised(b)))*
- All parts of the redundant block have size `BlockPartSizeBytes`

DF1e. Let *p* be *parts(serialised(b))*. Proposed block *b* and `Proposal` (or *signed proposal) prp* can be used as input to create a `CompactBlock` instance (or *compact block*) `cb`:

- `cb.blobs` is an array of `TxMetaData` populated from *b* as described in DF1b
- `cb.proposal` is *prp*
- `cb.last_length` is the length of the last part, i.e., *len(p[len(parts(serialised(b)))-1])*
- `cb.parts_hashes`
  - has length *2* len(parts(serialised(b)))*
  - `cb.parts_hashes[i]` the hash of *p[i]*, if *i < len(parts(serialised(b)))*
  - `cb.parts_hashes[i]` is the hash of *pp[i-len(parts(serialised(b)))]*, if i >= *len(parts(serialised(b)))*, where *pp* is *parity(serialised(b))*
- `cb.signature` contains the return value of *signBytes(*`cb2`*)*, where
  - `cb2`'s fields contain the same values as `cb`'s fields except `cb2.signature` which is set to `nil`
  - notice that it is not necessary to include `cb2.proposal` in the signature, as it is already signed by the same validator
- `cb.bp_hash` is the root of the merkle tree of the second half of `cb.parts_hashes[i]`
  - the second half is the indexes *i*, such that *len(parts(serialised(b))) <= i < 2* len(parts(serialised(b)))*

  - these are holding the hashes of the parts of *parity(serialised(b))*
  - N.B.: `cb.bp_hash` is the same to *parity(serialised(b))* as `cb.proposal.block_id.hash` is to *b*

DF1f. Proposed block *b*, part *p[i]*, and `CompactBlock` instance `cb` created from *b*, can be used as input to create a `RecoveryPart` instance `rp`:

- `rp.height` is `cb.proposal.height`
- `rp.round` is `cb.proposal.round`
- `rp.index` is *i*
- `rp.data` is *p[i]*
- `rp.proof` is a merkle proof of inclusion of `hash(rp.data)` as a leaf in the merkle tree whose root is `cb.proposal.block_id.hash`

DF1g. Let *pp* be *parity(serialised(b))*. Proposed block *b*, part *pp[i]*, and `CompactBlock` instance `cb` created from *b*, can be used as input to create a `RecoveryPart` instance `rp`:

- `rp.height` is `cb.proposal.height`
- `rp.round` is `cb.proposal.round`
- `rp.index` is *len(p) + i*
- `rp.data` is *pp[i]*
- `rp.proof` is a merkle proof of inclusion of `hash(rp.data)` as a leaf in the merkle tree whose root is `cb.bp_hash`

Data Flow 2: `CompactBlock` as input

Inputs => Outputs:

- `CompactBlock` => `HaveParts`
- `CompactBlock` => `WantParts`

DF2A. A compact block `cb` can be used as input to create a `HaveParts` instance `hp`.

- `hp.height` is `cb.proposal.height`
- `hp.round` is `cb.proposal.round`
- `hp.parts` is filled up as follows:
  - for *some* items `cbph` in `cb.parts_hashes` with index `i`, `hp.parts` contains an element `hpp` where:
  - `hpp.index` is `i`
  - `hpp.hash` is `cbph`

DF2B. A compact block `cb` can be used as input to create a `WantParts` instance `wp`.

- `wp.height` is `cb.proposal.height`
- `wp.round` is `cb.proposal.round`
- `wp.parts` is configured to flag the block parts of `cb` wanted
- `wp.prove` is `true` if the sender of `wp` requires a merkle proof in the recovery parts it is requesting with `wp`

Basic data validity definitions

Definition BD1: A hash *h* is valid iff *h*'s length is 32 (*sha256* size)

Notice that an empty hash is not valid

Data validity definitions

Definition DD1a: Let *b* be a proposed block. Let *p* be *parts(serialised(b))*. We say that part *p[i]*, *is a valid block part* iff:

- when 0 <= i < *len(parts(serialised(b)))-1,* we have that *len(p[i])* = `BlockPartSizeBytes`
- when i = *len(parts(serialised(b)))-1,* we have that *len(p[i])* <= `BlockPartSizeBytes`

Definition DD1b: Let *b* be a proposed block. Let *p* be *parity(serialised(b))*. We say that part *p[i]*, *is a valid block part* if *len(p[i])* = `BlockPartSizeBytes`

Definition DD2a: A `TxMetaData` instance `tm` is well formed iff

- `tm.hash` is a valid hash (BD1)
- `tm.start` < `tm.end`

Definition DD2b: We say that an instance `cb` of `CompactBlock` *is valid* iff

- The signature of `cb.Proposal.signature` is valid w.r.t. `cb.Proposal` — N.B. this signature verification logic is pre-existing in vanilla CometBFT
- `cb.signature` is valid according to the signed bytes as described in the previous section
- `cb.bp_hash` is a valid hash (BD1) and equal to the merkle root of the second half of the `cb.parts_hashed`
- `cb.last_length` <= `BlockPartSizeBytes`
- for every element `tm` in `cb.blobs` with index `i`
  - `tm` is well formed (DD2a)
  - there does not exist an element `tm2` in `cb.blobs` with index `j` such that
    - → `i` $\neq$ `j`, and
    - → either of these is true
      - · `tm.start` <= `tm2.start` <= `tm.end`
      - · `tm.start` <= `tm2.end` <= `tm.end`
- `cb.proposal.block_id` is a valid `Proposal` according to the pre-existing — vanilla CometBFT — validation rules. In addition, `BlockID.PartSetHeader.Hash` is equal to the merkle root of the first half of the `cb._parts_hashes`
- for every element `cbph` in `cb.parts_hashes`
  - `cbph` is a valid hash (BD1)

Observation (DD3): Given an instance `cb` of `CompactBlock`, it is not possible to validate elements of `cb.blobs` without access to

- either the entire proposed block
- or a list of contiguous `TxMetaData` elements that cover the offsets of a block part

We summarized this observation as an information finding ***TxMetaData cannot be validated before block is complete.***

Definition DD4: We say that an instance `rp` of `RecoveryPart` *is linked to* an instance `cb` of `CompactBlock` iff

- `rp.height = cb.proposal.height`
- `rp.round = cb.proposal.round`

Definition DD5: We say that an instance `rp` of `RecoveryPart` *is valid* iff

- `rp` is linked to an instance `cb` of `CompactBlock` (DD4)
- `cb` is valid (DD2b)
- `rp.index` < len(`cb.parts_hashes`)
- `cb.parts_hashes[rp.index] = hash(rp.data)`

- if `rp.index` < len(`cb.parts_hashes`) / 2
  - `rp.data` is a valid block part of *parts(serialised(b))* (DD1a)
  - `rp.proof`, together with `cb.parts_hashes[rp.index]` verify to the `cb.proposal.block_id.hash` as the merkle root
- if `rp.index` >= len(`cb.parts_hashes`) / 2
  - `rp.data` is a valid block part of *parity(serialised(b))* (DD1b)
  - `rp.proof`, together with `cb.parts_hashes[rp.index]` verify to the `cb.bp_hash` as the merkle root

Definition DD6: We say that an instance `hp` of `HaveParts` *is linked to* an instance `cb` of `CompactBlock` iff

- `hp.height = cb.proposal.height`
- `hp.round = cb.proposal.round`

Definition DD7: We say that an instance `hp` of `HaveParts` *is valid* iff

- `hp` is linked to an instance `cb` of `CompactBlock` (DD6)
- `cb`is valid (DD2b)
- for every item `hpp` in `hp.parts`
  - `hpp.hash` is a valid hash (BD1)
- for every item `hpp` in `hp.parts` with index `i`
  - there does not exist an item `hpp2` in `hp.parts` with index `j` such that
    - → `i` $\neq$ `j`, and
    - → `hpp.index` = `hpp2.index`
- for every item `hpp` in `hp.parts`
  - `hpp.index` < len(`cb.parts_hashes`), and
  - `hpp.hash` = `cb.parts_hashes[hpp.index]`

Definition DD8: We say that an instance `wp` of `WantParts` *is linked to* an instance `cb` of `CompactBlock` iff

- `wp.height = cb.proposal.height`
- `wp.round = cb.proposal.round`

Definition DD9: We say that an instance `wp` of `WantParts` *is valid* iff

- `wp` is linked to an instance `cb` of `CompactBlock` (DD8)
- `cb`is valid (DD2b)
- the size of `wp.parts` is equal to *len(*`cb.parts_hashes`*)*
- at least one bit of `wp.parts` is set

Data validation properties

Property DV1: Upon receiving an instance `cb` of `CompactBlock` from the network, if `cb` is not valid (DD2b), then `cb` is dropped

Description: The first thing a node MUST do upon reception of a compact block is validating it.

Property DV2a: Upon receiving an instance `rp` of `RecoveryPart` from the network, if `rp` is not linked to a compact block `cb` (DD4) that has previously been received by the local node, then `rp` is dropped

Description: We MUST NOT accept recovery parts from the network if we don't have the corresponding compact block to verify them. We don't even want to buffer them to prevent potential OOM problems.

Property DV2b: Upon receiving an instance `rp` of `RecoveryPart` from the network, if `rp` is linked to a compact block `cb` (DD4) that the local node previously received, and `rp` is not valid (DD5), then `rp` is dropped.

Description: The first thing a node MUST do upon reception of a recovery part is validating it.

Property DV3a: Upon receiving an instance `hp` of `HaveParts` from the network, if `hp` is not linked to a compact block `cb` (DD6) that has previously been received by the local node, then `hp` is dropped

Description: Similarly to DV2a, we MUST NOT accept “have parts” messages from the network if we don’t have the corresponding compact block to verify them.

Property DV3b: Upon receiving an instance `hp` of `HaveParts` from the network, if `hp` is linked to a compact block `cb` (DD6) that the local node previously received, and `hp` is not valid (DD7), then `hp` is dropped.

Description: The first thing a node MUST do upon reception of a “have parts” message is validating it.

Property DV4a: Upon receiving an instance `wp` of `WantParts` from the network, if `wp` is not linked to a compact block `cb` (DD8) that has previously been received (or created) by the local node, then `wp` is dropped

Description: We MUST NOT accept “want parts” messages from the network if we don’t have the corresponding compact block, as we won’t be able to respond with the data requested.

Property DV4b: Upon receiving an instance `wp` of `WantParts` from the network, if `wp` is linked to a compact block `cb` (DD8) that the local node previously received (or created locally), and `wp` is not valid (DD9), then `wp` is dropped.

Description: The first thing a node MUST do upon reception of a “want parts” message is validating it.

Property DV4c: Upon receiving an instance `wp` of `WantParts` from a peer `p` where `wp` is linked to a previously received compact block `cb` (DD8), and `wp` is valid, if any of the conditions below do not hold, then `wp` is dropped.

- the local node has previously sent `p` an instance `hp` of `HaveParts` linked to `cb` (DD6)
- if `wp`’s field `wp.parts` contains the `i`th bit set, then there is an element `hpp` in `hp.parts` such that `hpp.index = i`

Description: The first half makes sure the local node does not react to spurious request for parts. The second half makes sure the local node dismisses any request from a peer for a part that the local node did not advertise it has to that peer.

Property DV5: Let *b* be a proposed block. We have that *len(parts(serialised(b)))* = len(*parity(serialised(b)) and len(parts(serialized(b))) > 0*

Description: We MUST NOT have a block that, when serialised, produces no parts or no redundant parts, and the number of redundant parts MUST be equal to the number of original parts.

Algorithm description

This section is an informal description of the High Throughput Recovery algorithm, which we believe is a useful read before we delve into more formal properties in the next section.

One step

In a nutshell, this section explains the following basic sequence:

1. at time `t0`
   - `n1` sends `CompactBlock` to `n2`
   - `n1` sends `HaveParts` to `n2`

2. at time `t0 + delta`
   - `n2` sends `WantParts` to `n1`
3. at time `t0 + 2 delta`
   - `n1` sends a list of `RecoveryPart` instances to `n2`

Where node `n1` is directly connected to `n2`. Neither `n1` nor `n2` is the proposer of the block. For simplicity, we assume all nodes' mempools are empty.

Notice that, although this sequence does not include the proposer node `n`, the section's text needs to include references to `n` in some parts.

Sending compact block. When a proposer node `n` creates a proposed block *b*. It MUST create a compact block `cb` from *b*, and MUST send `cb` — shipped in a compact block message — to all of `n`'s peers.

Proposer sending `HaveParts`. Once `cb` is sent, `n` MUST send a `HaveParts` message `hp_i`, linked to `cb` (DD6), to each of its peers `p_i`. For each message `hp_i`, field `hp_i.parts` contains a — possibly different — list of `PartMetaData` instances whose indexes are interpreted as pointers to the elements of a subset `s_i` of *parts(serialised(b)) U parity(serialised(b))*. The rules governing which parts each subset `s_i` contains are described in the section "Load Balancing" below.

Part reconstruction and replying with `WantParts`. When a node `n2` receives a `HaveParts` message `hp` from another node `n1`, `n2` MUST previously have received a valid compact block `cb` from `n1`, to which `hp` is linked (DV3a). Then, `n2` validates `hp` against `cb` (DV3b), and checks which block parts referred to in `hp.parts` it can reconstruct from the transactions `n2` had previously received (i.e., from `n2`'s mempool reactor), and the data in `cb.blobs`. Additionally, `n2` tries to reconstruct missing parts from parity parts it may have already reconstructed. After this, if there are parts in `hp.parts` that cannot be reconstructed, `n2` sends a `WantParts` message `wp` linked to `cb` (DD8) back to `n1`. For every part in `hp.parts` that `n2` cannot reconstruct, `wp.parts` MUST have the corresponding bit set.

Replying with `RecoveryPart`. When a node `n1` receives a `WantParts` message `wp` from `n2`, `n1` MUST previously have sent a valid `HaveParts` message linked to a compact block `cb` to `n2` (DV4c). Additionally, `wp` should also be linked to `cb`. Here, we have two cases:

- If `n1` has the part requested, it ships it in a `RecoveryPart` instance and sends it back to `n2`.
- If `n1` does not have the part requested, it waits until it receives the part and then sends it as described in the previous bullet. The pipelining, explained in the following section, ensures that `n1` does eventually receive the part requested in normal conditions.

Multi-step & Pipelining

Schematically, this section deals with how compact blocks, `HaveParts`, and `WantParts` are propagated across nodes. In the steps below, node `n1` is directly connected to `n2`; node `n2` is directly connected to `n3`, and no other direct connections exist. None of those three nodes is the proposer of the block. For simplicity, we assume all nodes' mempool reactors don't have access to any transaction.

1. at time `t0`
   - `n2` receives `CompactBlock` from `n1`
   - `n2` receives `HaveParts` from `n1`
   - `n2` sends `WantParts` to `n1`
   - `n2` sends `CompactBlock` to `n3`
   - `n2` sends `HaveParts` to `n3`
2. at time `t0 + delta`
   - `n1` receives `WantParts` from `n2`
   - `n3` receives `CompactBlock` from `n2`
   - `n3` receives `HaveParts` from `n2`

- `n1` sends a list of `RecoveryPart` instances to `n2`
- `n3` sends `WantParts` to `n2`

3. at time `t0 + 2 delta`
   - `n2` receives `WantParts` from `n3`
   - `n2` receives a list of `RecoveryPart` instances from `n1`
   - `n2` sends a list of `RecoveryPart` instances to `n3`
4. at time `t0 + 3 delta`
   - `n3` receives a list of `RecoveryPart` instances from `n2`

Pipelining. Following from the previous section, given a node `n2` that receives a `HaveParts` message `hp` for the first time from `n1`, we have that, for every part `hpp` in `hp.parts`,

- (a) `n2` can reconstruct `hpp` from the mempool reactor's info and/or parity parts
- (b) otherwise, `n2` has set the corresponding bit in `wp.parts` in the `WantParts` message `n2` sent back to `n1`.

At this point, `n2` MUST signal to all its peers (except `n1`) that it has *all* parts in `hp.parts` (i.e., all parts `n1` told `n2` it has), even those of case (b) above. Thus, `n2` is advertising that it also has the parts it told `n1` it wants in `wp` although `n2` hasn't received them yet. This is called *pipelining* and helps reduce the algorithm's latency across the network.

Compact block and `HaveParts` propagation. Node `n2` MUST thus send `cb`, followed by a `HaveParts` message `hp2_i` linked to `cb` (DD6) to each of its peers `p2_i`. The parts advertised in `hp2_i.parts` of each `hp2_i` message MUST be the same. This rule differs substantially from the case when the sending node is the block proposer — described in sections "One step" above, and "Load Balancing" below — where each peer receives a different subset of the parts in the `hp_i` message.

Notice that, according to these rules, `n2` claims to its peers (except `n1`) that it has the same parts `n1` claims to have. Let's denote those parts `hp.parts`. This property goes back — transitively — all the way up to the first node that received `hp.parts` directly from the proposer node `n`. Remember that the rules for proposer node `n` are different.

Reducing duplicates and stopping propagation. To reduce the number of duplicates, node `n2`

- does not send `cb` to peers that previously sent `cb` to `n2`
- does not send an `hp_i` message to a peer that sent `n2` the same `hp_i` message.

Compact block and `HaveParts` flooding. At this point, we can observe that compact blocks are broadcast to the whole network using a flood mechanism similar to the way CometBFT `v1.0.x` broadcasts unconfirmed transactions. However, unlike transactions, a compact block has a small, predictable size.

As for `HaveParts`, they are flooded but in a different way. As the proposer node `n` can send `HaveParts` messages `hp_i` to its peers with different contents in `hp_i.parts`, what is actually flooded — excluding `n` — is each of the messages `hp_i`, as long as all of them have different contents in `hp_i.parts`. Just as compact blocks, `HaveParts` messages have a small, predictable size.

Load Balancing

At high level, this section deals with how a proposer node load balances the different `HaveParts` across its peers. In the steps below, proposer node `n` and nodes `n2` and `n3` are fully connected. For simplicity, we assume all nodes' mempool reactors don't have access to any transaction.

1. at time `t0`
   - `n` sends `CompactBlock` to all its peers (`n2` and `n3`)
     - The compact block contains the hashes of parts `p0` and `p1`
   - `n` sends `HaveParts` to `n2`, with `p0`

- `n` sends `HaveParts` to `n3`, with `p1`

2. at time `t0 + delta`
   - `n2` receives `CompactBlock` from `n`
   - `n2` receives `HaveParts` from `n`, with `p0`
   - `n3` receives `CompactBlock` from `n`
   - `n3` receives `HaveParts` from `n`, with `p1`
   - `n2` sends `WantParts` to `n`, with `p0`
   - `n2` sends `HaveParts` to `n3`, with `p0`
   - `n3` sends `WantParts` to `n`, with `p1`
   - `n3` sends `HaveParts` to `n2`, with `p1`
3. at time `t0 + 2 delta`
   - `n2` receives `RecoveryPart` from `n`, with `p0`
   - `n3` receives `RecoveryPart` from `n`, with `p1`
   - `n3` sends `WantParts` to `n2`, with `p0`
   - `n2` sends `WantParts` to `n3`, with `p1`
   - `n2` sends `RecoveryPart` to `n3`, with `p0`
   - `n3` sends `RecoveryPart` to `n2`, with `p1`
4. at time `t0 + 3 delta`
   - `n3` receives `RecoveryPart` from `n2`, with `p0`
   - `n2` receives `RecoveryPart` from `n3`, with `p1`

At this point, both `n2` and `n3` have `p0` and `p1`, and the proposer node `n` only sent one part to each of its peers.

Let us assume that, given a proposed block, the proposer node `n` sends to all its peers `p_i` the following

- first, the corresponding compact block
- then, `HaveParts` messages `hp_i`, where `hp_i.parts` for all `i` contains all parts in *parts(serialised(b)) U parity(serialised(b))*

In this base routing policy, whereby the `n` sends all parts to all its peers, the `n` itself becomes the root of the tree of all `HaveParts` it just sent.

According to the algorithm described in section "One Step", every peer `p_i` will then send back to `n` a `WantParts` message `wp`, where the `wp.parts` bitmask has all bits set. This risks saturating `n`'s egress bandwidth, since `n` must upload every part to all its peers.

To reduce the possibility of running into this problem, rather than the base routing policy, we use the following one:

- Let `k` be the number of `n`'s peers
- Let *S* be the set of all parts of the proposed block, i.e., *S = parts(serialised(b)) U parity(serialised(b))*
- Proposer node `n` partitions set *S*
  - into *k* disjoint subsets *S_i* (see section "Algorithmic Definitions" below for more detail)
  - if | *S* | > *k*, then sets *S_i* are singletons and not disjoint: some *S_i* may have the same part, as long as every part is in one set *S_i*
- Additionally, the size of all sets *S_i* MUST be well balanced: the size difference of any two sets *S_i* MUST NOT be more than 1.
  - This MAY be relaxed in more advanced policies where the bandwidth to each of `n`'s peers is measured and monitored. However, we ignore these optimisations in this threat model for simplicity

Once `n` has partitioned *S*, for every peer `p_i`, `n` creates a `HaveParts` message `hp_i`, ships *S_i* in `hp_i.parts`, and sends it to `p_i`.

Now, rather than just one broadcast tree with root `n`, this results in *k* broadcast trees, one for each set of parts *S_i*, with root `p_i`. Additionally, `n` cannot be part of any of those *k* broadcast trees.

Assumption. In order to come up with a complete model before the threat analysis starts, we will henceforth make the following assumption:

- Let N be the set of all nodes in the network
- ∀ *i*: 0 <= *i* < *k*, ∀ `n2` ∈ N \ {`n`}; there is at least one path from `p_i` to `n2` that
  - only contains honest nodes
  - does not contain `n`

Informally, we assume that all of `n`'s peers are able to reach the whole network without needing to go through `n`.

Algorithmic definitions

Definition AD1: We say a `CompactBlock` instance `cb` is derived from a proposed block *b* and a signed proposal *prp* iff it follows the rules defined in DF1e.

Definition AD2: Let `n` be a proposer node proposing block *b*. Let *k* be the number of `n`'s peers. Let *S* be the set of all parts of the proposed block, i.e. *S = parts(serialised(b)) U parity(serialised(b))*. We say proposer node `n` *distributes* set *S* into *k* subsets *S_i* iff:

- ∄ *p* ∈ *S*; such that *p* ∉ *S_i*, ∀ *i*: *0* <= *i* < *k*
- ∄ *e*, ∄ *i*: *0* <= *i* < *k*; such that *e* ∈ *S_i* and *e* ∉ *S*

Definition AD3: We say a `HaveParts` message `hp` contains a `WantParts` message `wp` iff

- `wp` is linked to a `CompactBlock` `cb` (DD8)
- `hp` is also linked to `cb` (DD6)
- for every bit set in `wp.parts`, with index *i*
  - ∃ `pmd` ∈ `hp.parts`; such that `pmd.index` = *i*

Definition AD4: Let *b* be the proposed block for height *h* and round *r*. Let p be *parts(serialised(b)) U parity(serialised(b))*. Every node `n_i` defines set *outstanding(*`n_i`*, h, r, j)* as the set of `n_i`'s peers that requested part *p[j]* while `n_i` does not have — or cannot reconstruct — *p[j]*.

Definition AD5: We say a `WantParts` message `wp` contains a `RecoveryPart` message `rp` iff

- `rp` is linked to a `CompactBlock` `cb` (DD4)
- `wp` is also linked to `cb` (DD8)
- Let *i* be the value in the `rp.index`
  - then, the *i*-th bit in `wp.parts` is set

Definition AD6: Let *b* be a proposed block. Let `cb` be an instance of `CompactBlock` derived from *b* (AD1). We say a `WantParts` message `wp` requests part *p* iff `wp` is linked to `cb` and `wp.parts` contains the *i*-th bit set, where *i* is the index of the element of `cb.parts_hashes` that contains *hash(p)*.

Algorithmic properties

In this section, we formally describe the rules that govern the High Throughput Recovery algorithm, which honest nodes are to follow.

The structure mimics the one in the “Algorithm Description” section.

One Step

Property AL1: A proposer node `n` of proposed block *b* sends `CompactBlock` `cb` derived from *b* (AD1) to all its peers.

Description: The Proposer node needs to make sure the compact block it just created reaches all nodes in the network. As the compact blocks' size is small and predictable, we can afford to send to to all peers.

Property AL2: A node `n1` sends a peer `p` a `HaveParts` message `hp` linked to a `CompactBlock` `cb` (DD6) only after having sent `cb`.

Description: This ensures that `p` does not drop `cb` (DV3a).

Property AL3: If a node `n1` receives a `HaveParts` message `hp` for the first time, and does not drop it (DV3a, DV3b), and cannot reconstruct one part *p* referred to in `hp.parts` locally, `n1` sends a `WantParts` message `wp` requesting *p* to the sender of `hp` (AD6).

Description: This allows `n1` to retrieve all parts it cannot reconstruct, and for which it also advertises to its peers that it has (or will have) via a `HaveParts` message.

Property AL4: A node `n1` does not send a `WantParts` message `wp` to a peer `p` if `n1` did not previously receive from p a `HaveParts` message `hp` that contains `wp` (AD3).

Description: This property specifies part of the one-step correct behaviour that honest nodes have to follow.

Property AL5: If a node `n1` receives a `WantParts` message `wp` from peer `p`, and `n1` does not drop `wp` because of DV4a, DV4b, or DV4c, and `n1` did not previously send `p` a `HaveParts` message `hp` that contains `wp` (AD3), then `n1` drops `wp`.

Description: Nodes should not deal with spurious `WantParts` messages.

Property AL6: If a node `n1` receives a `WantParts` message `wp` from peer `p`, and `n1` does not drop `wp` because of DV4a, DV4b, DV4c, or AL5

- let *p* be *parts(serialised(b)) U parity(serialised(b))*
- let `cb` be the `CompactBlock` to which `wp` is linked (DD8)
- for every bit set in `wp.parts`, with index *i*
  - if `n1` has — or can reconstruct — *p[i]*, then `n1` sends *p[i]* in a `RecoveryPart` message `rp` , populated from `cb` and *p[i]* according to DF1f or DF1g.
  - if `n1` does not have — and cannot reconstruct — *p[i]*, then `n1` adds (AD4) `p` to set *outstanding(*`n1`*,* `wp.height`*,* `wp.round`*, i)*

Description: Upon receiving a `WantParts` message, a node MUST send back the part if it has it, or keep track of the request — the goal of sets *outstanding,* to send it when the node finally receives the part.

Property AL7: A node `n1` sends a `RecoveryPart` message `rp` to a peer `p` iff `n1` previously received a `WantParts` message `wp` from `p` such that `wp` contains `rp` (AD5).

Description: Correct processes MUST NOT send spurious `RecoveryPart` messages, and they MUST send `RecoveryPart` messages when requested.

Property AL8: If a node `n1` receives a `RecoveryPart` message `rp` from peer `p`, and `n1` does not drop `rp` because of DV2a, DV2b, or AL14 (below), and `n1` did not previously send `p` a `WantParts` message `wp` that contains `rp` (AD5), then `n1` drops `rp`.

Description: Nodes should not deal with spurious `RecoveryPart` messages.

Multi-Step

Property AL9: If a node `n1` receives a `CompactBlock` message `cb` for the first time, and does not drop it because of DV1, `n1` sends `cb` to all its peers, except the peer from which `n1` received `cb`.

Description: This property is the core mechanism whereby compact blocks are flooded in the network.

Property AL10: If a node `n1` receives a `HaveParts` valid message `hp`, `n1` sends a `HaveParts` message `hp'` to all its peers (unless `hp'` is empty), except the peer that `n1` received `hp` from, where `hp'` contains all parts from `hp` for which `n1` has not sent a `HaveParts` message before.

Description: All nodes, except the proposer, relay those parts of `HaveParts` messages they have not sent yet, which generates the broadcast tree for the parts contained in `hp.parts`.

Property AL11: If a node `n1` receives a `RecoveryPart` message `rp`, and `n1` does not drop `rp` because of DV2a, DV2b, AL8, or AL14 (below), `n1` sends `rp` to all peers in set *outstanding(*`n1`*,* `rp.height`*,* `rp.round`*,* `rp.index`*)*, and resets the contents of the set to the empty set.

Description: Upon receiving a recovery part, a node MUST relay it to all those that asked for it previously, apart from using it locally to reconstruct the whole block.

Load Balancing

Property AL12: Let `n` be a proposer node proposing block *b*. Let `cb` be the `CompactBlock` derived from *b* (AD1). Let `p_i` be one of the proposer node `n`'s *k* peers `p_i`. Proposer node `n` distributes (AD2) set *S* into subsets *S_i*, if | *S* | < *k*, and sends a `HaveParts` message `hp_i`, with *S_i* in `hp_i.parts` to `p_i`.

Description: As stated in section "Load Balancing", `n` does not send the same part to more than one of its peers (unless there are more peers than parts), to avoid having to upload it more than once.

General Properties

Property AL13: A node `n1` does not send a message `m` more than once to a peer `p`. This property applies for the cases where `m` is an instance of `CompactBlock`, `HaveParts`, `WantParts`, or `RecoveryPart`.

Description: The algorithm does not need the same message sent between `n1` and peer `p` more than once, therefore, `n1` MUST NOT do that to avoid flooding the network with useless messages.

Property AL14: If a node `n1` receives a message `m` from `p`, and `n1` previously received `m` from `p`, then `n1` drops `m`. Message `m` can be an instance of `CompactBlock`, `HaveParts`, `WantParts`, or `RecoveryPart`.

Description: This property is complementary to AL13 and prevents a node from being overwhelmed with many copies of the same message. Notice that this property is restricted to the same part of nodes `n1` and `p`, since we cannot avoid cases where a node receives the same message `m` from several peers. Also, `n1` and `p` may decide to send `m` to each other at the same time (crossed messages), and this should not be considered incorrect behaviour.

Property AL15: If a node `n1` drops a message received from peer `p` because of `p`'s misbehaviour — i.e. any of these properties held: DV1, DV2a, DV2b, DV3a, DV3b, DV4a, DV4b, DV4c, AL5, AL8, AL14 — `n1` disconnects from `p` and downgrades `p`'s score in `n1`'s address book.

Description: This allows a node to get rid of misbehaving nodes, at least for a while.

Property AL16: A node `n1` sends message `m` iff `m` is valid. Where `m` can be an instance of `CompactBlock` (DD2b), `HaveParts` (DD7), `WantParts` (DD9), or `RecoveryPart` (DD5).

Description: Obviously, correct nodes MUST NOT send garbage messages.

Non-property AL17: Even if a node `n1` and all its peers respect all properties in this threat model, once `n1` is able to reconstruct the proposed block *b*, `n1` may find out that *b* is invalid. This can happen if proposer node `n` of block *b* is not honest.

Description: The whole system of messages and properties in this model cannot prevent the initially proposed block from being incorrect. Hence, property AL18.

Property AL18: Once a node `n1` is able to reconstruct a proposed block *b*, `n1` validates the block according to the block validation rules pre-existing in CometBFT.

Threats

This section lists all the threats that can be derived from the list of properties specified in the section "Properties".

Some threats arise from violations of a definition, which often involve multiple conditions. To keep the list of threats concise, rather than splitting each condition into separate threats, we grouped them under a single threat definition. During the analysis, the security analyst considered each condition individually within the same threat.

It is important to clarify the usage of "or" in some threat definitions. The meaning of "or" is that the threat is, actually, a combination of several similar threats. The analyst has examined each "branch" of the "or" as though it was one threat on its own. This helps keep this model more compact.

Data validation threats

DVT1: A (non-proposer) node `n1` receives from its peer `p` a `CompactBlock` instance `cb`, `cb` is not valid (DD2b). Node `n1` does not drop `cb`, or does not disconnect from `p`.

- Description: A node is not rejecting a `CompactBlock` that has a problem. Each of the conditions in DD2b is analysed as part of this threat.
- Related properties: DV1, AL15.
- Inspection results: The threat holds as there is no complete validation of received blocks and the connection is not always dropped:
  - Upon reception of a compact block, only the proposal is validated by using the validation function from CometBFT: In `validateCompactBlock` the function that is stored in the variable `proposalValidator` is called, this variable is set in `NewNodeWithContext` to call `consensusState.ValidateProposal(proposal)` [OK]
  - There is no verification of the signature (Finding ***Discrepancy between the implementation and specification***)
  - The proof cache is generated and the merkle root of the parts and the parity parts are compared to `cb.Proposal.BlockID.PartSetHeader.Hash` and `cb.BpHash` [OK]
  - If there is an error in the proofs, the proposal cache entry is deleted [OK]
  - The integrity of blobs (`TxMetaData`) was check in `CompactBlock.ValidateBasic()`
  - `last_length` is not checked. This can lead to a panic when decoding the block. See Finding ***Not validating last part length of CompactBlock leads to a panic while decoding***.
  - A connection is not dropped if the proposal validation fails (Finding ***The specified disconnection rules were not implemented***), but if there are errors in the root hash of the block parts and parity parts wrt. `cb.Proposal.BlockID.PartSetHeader.Hash` and `cb.BpHash`
  - Additionally, note the findings ***The CompactBlock validation doesn't check whether the proposal is for the right height and round***, and ***`if !blockProp.started.Load()` check missing from the `handleCompactBock` function***.

DVT2: A (non-proposer) node `n1` receives from its peer `p` a `RecoveryPart` instance `rp`, and `rp` either (a) is not linked to any `CompactBlock` instance `cb` (i.e. `rp` is malformed), or (b) is not linked to a `CompactBlock` instance `cb` (DD4) that `n1` has created or received. Node `n1` does not drop `rp`, or does not disconnect from `p`.

- Description: A node is not rejecting a `RecoveryPart` message it cannot verify.
- Related properties: DV2a, AL15.

- Inspection results: There is no risk of this threat occurring.`RecoveryPart` messages are received in `Reactor.ReceiveEnvelope()` and handled via the `handleRecoveryPart` method. The height and round of the `RecoveryPart` message are used to retrieve the `proposalData` from the `PorposalCache`, which contains the previously received `CompactBlocks`. If no `CompactBlock` is found, the `handleRecoveryPart` method returns, and the message is dropped.
  - Note that the node doesn't disconnect from the peer that sends the message (code ref↗), i.e., property AL15 doesn't hold. This is consistent with the notes received from the client, i.e., *Relying on per peer bandwidth constraints*.
  - Note that the threat cannot occur during the catchup mechanism either, as the node creates a `CompactBlock` instance before sending requests for parts (code ref↗) and drops the any recovery parts that are not linked to a compact block.

DVT3: A (non-proposer) node `n1` receives from its peer `p` a `RecoveryPart` instance `rp`, `rp` is linked to a `CompactBlock` instance `cb` (DD4) that `n1` has created or received, and `rp` is not valid (DD5). Node `n1` does not drop `rp`, or does not disconnect from `p`.

- Description: A node is not rejecting a `RecoveryPart` message that does not verify against the `CompactBlock` it is linked with.
- Related properties: DV2b, AL15.
- Inspection results: There is no risk of this threat occurring. `n1` drops `rp` if it's not linked to a `cb` that `n1` created or received (see argument for DVT2). Furthermore, the following arguments show that `n1` validates `rp` before handling it (as defined in DD5).
  - `rp` is linked to an instance `cb` of `CompactBlock` (DD4)
    - → See argument for DVT2.
  - `cb` is valid (DD2b)
    - → See argument for DVT1. Note though that if `cb` is created during the catchup mechanism (code ref↗), DD2b doesn't hold. However this doesn't have any practical impact in the code.
  - `rp.index` < len(`cb.parts_hashes`)
    - → This is ensured by the check in `AddPartWithoutProof` (code ref↗), together with the index adjustment in `CombinedPartSet.AddPart` (code ref↗). Note that this↗ check ensures that `len(cb.parts_hashes) = 2 * len(original parts)`.
    - → Doesn't apply for catchup mechanism as `cb.parts_hashes` is empty. Note that the above check doesn't happen as the `Proofs` method is never called.
  - `cb.parts_hashes[rp.index] = hash(rp.data)`
    - → This is ensured indirectly by verifying the merkle proof in `PartSet.AddPart` (code ref↗). Note that the proofs were generated when receiving a new `CompactBlock` (code ref↗).
    - → Doesn't apply for catchup mechanism as `cb.parts_hashes` is empty.
  - if `rp.index` < len(`cb.parts_hashes`) / 2
    - → `rp.data` is a valid block part of *parts(serialised(b))* (DD1a)
      - · This is ensured under the assumption of an honest proposer that correctly splits a block into its parts (code ref↗). Note that `n1` verifies this by verifying `rp.proof` (see below) and eventually decoding all the parts (code ref↗). In addition, property AL18 ensures that `n1` is able to reconstruct the proposed block `b`.
    - → `rp.proof`, together with `cb.parts_hashes[rp.index]` verify to the `cb.proposal.block_id.hash` as the merkle root
      - · This is ensured via the following path: `Reactor.handleRecoveryPart` (code ref↗) -> `CombinedPartSet.AddPart` (code ref↗) -> `PartSet.AddPart` (code ref↗)
      - · Note that `CombinedPartSet.original.hash` is set to `cb.proposal.block_id.hash` when a new `CompactBlock` is added via `AddProposal` (code ref↗)

        - Also, the proof is retrieved from the proof cache created when handling the corresponding `CompactBlock`
        - Doesn't apply for catchup mechanism as `cb.parts_hashes` is empty.
  - if `rp.index` >= len(`cb.parts_hashes`) / 2
    - → `rp.data` is a valid block part of *parity(serialised(b))* (DD1b)
      - This is ensured under the assumption of an honest proposer that correctly splits a block into its parts (code ref↗) and encodes it using the `reedsolomon` erasure coding (code ref↗). Note that `n1` verifies this by verifying `rp.proof` (see below) and eventually decoding all the parts (code ref↗). In addition, property AL18 ensures that `n1` is able to reconstruct the proposed block `b`.
    - → `rp.proof`, together with `cb.parts_hashes[rp.index]` verify to the `cb.bp_hash` as the merkle root
      - This is ensured via the following path: `Reactor.handleRecoveryPart` (code ref↗) -> `CombinedPartSet.AddPart` (code ref↗) -> `PartSet.AddPart` (code ref↗)
      - Note that `CombinedPartSet.parity.hash` is set to `cb.bp_hash` when a new `CompactBlock` is added via `AddProposal` (code ref↗)
      - Doesn't apply for catchup mechanism as `cb.parts_hashes` is empty.

DVT4: A (non-proposer) node `n1` receives from its peer `p` a `HaveParts` instance `hp`, and `hp` either (a) is not linked to any `CompactBlock` instance `cb` (i.e., `hp` is malformed), or (b) is not linked to a `CompactBlock` instance `cb` (DD6) that `n1` has created or received. Node `n1` does not drop `hp`, or does not disconnect from `p`.

- Description: A node is not rejecting a `HaveParts` message it cannot verify.
- Related properties: DV3a, AL15.
- Inspection results:

If the compact block cannot be retrieved for the specified height and round, the `handleHaves` function returns nil (code ref↗). The retrieved compact block is validated before storing (code ref↗), as discussed in DVT1. If `HaveParts` isn't linked to any `CompactBlock`, through matching height and round values, the function returns early with just a debug log.

The message is dropped implicitly with the return, as other messages continue to be processed. Note the finding ***The specified disconnection rules were not implemented***.

DVT5: A (non-proposer) node `n1` receives from its peer `p` a `HaveParts` instance `hp`, `hp` is linked to a `CompactBlock` instance `cb` (DD6) that `n1` has created or received, and `hp` is not valid (DD7). Node `n1` does not drop `hp`, or does not disconnect from `p`.

- Description: A node is not rejecting a `HaveParts` message that does not verify against the `CompactBlock` it is linked with.
- Related properties: DV3b, AL15.
- Inspection results:

The function `ValidateBasic` checks if the `HaveParts` is nil or empty or if any of the parts is invalid (code ref↗) by checking the hash (code ref↗).

The code does not explicitly check for duplicate indices in `hp.parts`. If `HaveParts` has n entries with duplicate indices and `CompactBlock` has n parts, then some indices in `CompactBlock` could be unmatched, meaning `HaveParts` message can be incomplete/invalid (code ref↗). On the other hand, `GetTrueIndices` ensures each index only appears once in the request generation, so even if duplicate indices pass initial validation, they don't cause security issues in the request handling (code ref↗). The only waste is in the initial processing of duplicate indices in the loop (code ref↗).

`ValidatePartHashes` validates that each index is within bounds of the `CompactBlock`'s `PartsHashes` array and that each hash matches the index- corresponding hash in `cb.PartsHashes` (code ref↗). If the validation fails, the peer is dropped, and the processing stops (code ref↗).

DVT6: A (non-proposer) node `n1` receives from its peer `p` a `WantParts` instance `wp`, and `wp` either (a) is not linked to any `CompactBlock` instance `cb` (i.e. `wp` is malformed), or (b) is not linked to a `CompactBlock` instance `cb` (DD8) that `n1` has created or received. Node `n1` does not drop `wp`, or does not disconnect from `p`.

- Description: A node is not rejecting a `WantParts` message it cannot verify.
- Related properties: DV4a, AL15.
- Inspection results:

The threat doesn't hold for the regular flow. The `getAllState` function first checks whether the node is in catch-up mode and whether the provided height and round are less than the consensus height and round (code ref↗). Later, the height is compared to the `currentHeight`, and if the node is not in catch-up mode, this check will fail, causing the function to return `false` and set `hasStored` to `nil` (code ref↗). This behavior ensures that when the node is not in catch-up mode, `getAllState` will only return `true` for `has` if it successfully retrieves the compact block (code ref↗). The `handleWants` method then checks the value of `has`, and if it's `false`, the method returns early (code ref↗). If the `WantParts` message was part of the regular flow, and not catchup, it will be linked to a compact block instance. The retrieved compact block is validated before being stored (code ref↗), as discussed in DVT1.

The catch-up flag is determined by the `wants.Prove` field. If `wants.Prove` is set to `true`, the `handleWants` function treats the message as originating from the catch-up mechanism. In this case, the `hasStored` variable in the `getAllState` function may contain a block retrieved from the store (code ref↗). This flow is analyzed in the threat model for the catch-up mechanism.

DVT7: A (non-proposer) node `n1` receives from its peer `p` a `WantParts` instance `wp`, `wp` is linked to a `CompactBlock` instance `cb` (DD8) that `n1` has created or received, and `wp` is not valid (DD9). Node `n1` does not drop `wp`, or does not disconnect from `p`.

- Description: A node is not rejecting a `WantParts` message that does not verify against the `CompactBlock` it is linked with.
- Related properties: DV4b, AL15.
- Inspection results:

The threat holds. The `WantParts` message will be dropped if it isn't linked to the instance of the compact block outside the catchup mechanism, as described in the DVT6.

The function `handleWants` doesn't process the `WantParts` message if at least one bit of `wp.parts` is not set. If no bits are set, `canSend` will be empty, `GetTrueIndices` would return an empty slice, and the code inside the loop will never execute (code ref↗).

When handling `WantParts` messages intended for the regular flow, the `wp.prove` field is set to `false`. The `getAllState` function checks whether the node is currently in catch-up mode. If `wp.prove` is not set, it verifies that the provided height and round are relevant to the current consensus state (code ref↗). Otherwise, it assumes the `WantParts` message is part of the catch-up mechanism and confirms this by checking if the height is less than the `currentHeight` (code ref↗).

The code doesn't check the size of the `wp` message. The `And` operation (code ref↗) returns nil only if either `BitArray` is nil, otherwise, it creates a new `BitArray` with the minimum size of the two arrays and performs the and operation on the elements up to that minimum size (code ref↗). If a `WantParts` message with `wp.parts` larger than `len(cb.parts_hashes)` is handled, the `And` function will shorten that array, and handle wants will send parts within the bounds. However, the oversized wants will be stored in the `PeerState` as missing wants and won't be removed until the function `prune` is called (code ref↗). See the finding ***Unresolvable wants in PeerState***.

DVT8: A (non-proposer) node `n1` receives from its peer `p` a `WantParts` instance `wp`, `wp` is linked to a `CompactBlock` instance `cb` (DD8) that `n1` has created or received, `wp` is valid (DD9), and one of the conditions in specified in DV4c does not hold. Node `n1` does not drop `wp`, or does not disconnect from `p`.

- Description: A node `n1` is not rejecting a valid `WantParts` message from peer `p` that either

  - `n1` did not even advertise to `p` it has some parts of `cb` in a `HaveParts` message, or
  - `n1` did not advertise to `p` some of the parts `p` is requesting in `wp`.
- Related properties: DV4c, AL15.
- Inspection results:

The code doesn't implement the check whether the `n1` did not advertise to `p` some parts of `cb` in a `HaveParts` message, nor the check if the `n1` advertised to `p` parts `p` is requesting in `wp`. Additionally, a malicious peer could repeatedly send `WantParts` messages since there's no explicit limit on how many times they can request the same parts. However, as noted by clients previously to the audit, they are relying on peer-to-peer bandwidth constraints. "*The spec envisions an ideal protocol without per-peer bandwidth restrictions. This can exist, but requires all edge cases to be covered before disconnecting from peers can be added. Since the current protocol does not handle these cases safely, we must retain per-peer bandwidth restrictions to maintain security.*"

Note the findings ***Race condition in `HaveParts` processing during height transitions***, and ***The specified disconnection rules were not implemented***.

DVT9: A proposer node `n` proposes a block *b* that results in *k* parts after serialisation (i.e. *len(parts(serialised(b)))* = *k*), and in *k'* parity parts after generating the parity parts from the block parts (i.e., *len(parity(serialised(b))* = *k'*). We have that $k \neq k'$ or *k* = *0*.

- Description: A block with zero parts cannot exist, as an empty block has, at the very least, the block header to serialise. Also, the `reedsolomon` library used in step DF1d produces as many parity parts as block parts are provided to the library as input.
- Related properties: DV5.
- Inspection results: There is no risk of this threat occurring. First, the logic for creating original block parts is not modified by the code under audit. Second, the encoding logic ensures that the number of parity parts generated equals the number of original parts (code ref↗). Furthermore, if the number of original parts is zero, then the encoding will fail (code ref↗).

Algorithmic threats

ALT1: A proposer node `n` proposing block *b* does not correctly derive `CompactBlock` `cb` from *b* (AD1), or does not send `cb` to all its peers.

- Description: If `n`, which is the origin of all information related to *b* — including `cb` and any `HaveParts` message linked to `cb` (DD6) — does not properly create or propagate `cb` to all its peers, it is a major issue.

- Related properties: AL1.

- Inspection results:

  When a proposer wants to propose a block, it calls `Reactor.ProposeBlock` with the Proposal, the block as `PartSet` and the transactions metadata txs.

  - The parity blocks as `PartSet` are computed by the function `Encode()`
    - → The part set for the parity bits is correctly computed [OK]
  - The hashes of the block parts and parity parts are extracted in `extractHashes` as array of byte arrays and used to fill the `PartHashes` field of the `cb`, which is according to the specification [OK]
  - The proofs of the block parts and parity parts are extracted in `extractProofs` as array of Proof and set as the proof cache for CB [OK]
  - Call to `handleCompactBlock` with a correctly derived compact block
    - → call to `broadcastCompactBlock`: the `cb` is sent to all peers
    - → Note that a failure in sending does not lead to retransmission (Finding ***TrySendEnvelopeShim can silently fail in broadcastCompactBlock function***)

ALT2: A (non-proposer) node `n1` sends a peer `p` a `HaveParts` message `hp` linked to a `CompactBlock` `cb` (DD6) but does not send `cb`, or sends `cb` after `hp`, to `p`.

- Description: Nodes MUST send `cb` first, followed by any `HaveParts` message. In this threat, the order is reversed.
- Related properties: AL2.
- Inspection results:

The threat does not hold. The `HaveParts` message is sent in the `ProposeBlock` method (code ref↗) after `handleCompactBlock` is called, which saves the compact block locally and broadcasts it to the connected peers (code ref↗).

Additionally, when broadcasting `HaveParts` in the `recoverPartsFromMempool` function (code ref↗), the `handleCompactBlock` method sequence enforces the ordering. The function first broadcasts `CompactBlock` (code ref↗) and then starts the `recoverPartsFromMempool` in a go routine if the peer is not a proposer (code ref↗).

`SendWantsThenBroadcastHaves` is only called in response to receiving `HaveParts` messages from peers (code ref↗), when the `receivedHaves` channel is populated in the `handleHaves` function (code ref↗). The `handleHaves` function checks the existence of the compact block and validates the part hashes (code ref↗).

The `CompactBlock` can be added to the cache through the `AddProposal` function in `handleCompactBlock` (code ref↗) or `AddCommitment` during catch-up (code ref↗). Consequently, having a non-nil `CompactBlock` in `getAllState` is not equivalent to having broadcast it.

However, `handleHaves` would log an error and return early for a `CompactBlock` created by `AddCommitment`, since the created `cb` only has height and round in its `Proposal` (code ref↗), and doesn't have `PartsHashes` set. In this case, the `ValidatePartHashes` check wouldn't pass (code ref↗).

`HaveParts` messages are also sent in `handleRecoveryPart` when parts are decoded (code ref↗). If a `CompactBlock` is created during catchup in `AddCommitment`, the `CombinedPartSet` is created with the `catchup` variable set to `true` (code ref↗). When `handleRecoveryPart` gets the parts through `getAllState`, it gets this same `CombinedPartSet` that was previously created (code ref↗). The `CanDecode` function will then return false, since the `!cps.catchup` condition will always be false (code ref↗). Otherwise, if the `CompactBlock` is added to the cache through the `AddProposal` function in `handleCompactBlock`, and it is broadcasted (code ref↗).

ALT3: A (non-proposer) node `n1` receives a valid `HaveParts` message `hp` from `p1`, and cannot reconstruct one part *p* referred to in `hp.parts` locally. Node `n1` relays `hp` to another peer `p2`, but does not send a `WantParts` message `wp` to `p1`, where `wp` requests for *p* (AD6).

- Description: If a node `n1` does not have a part, `n1` MUST request it, as it also MUST advertise to other peers it *has* that part. In this threat, `n1` is telling `p2` it has *p*, but it does not, and it does not do its due diligence of trying to obtain it.
- Related properties: AL3.
- Inspection results:

The threat holds. See the ***HaveParts broadcast despite the failed WantParts send*** finding.

ALT4: A (non-proposer) node `n1` sends a `WantParts` message `wp` to a peer `p`. Peer `p` has not sent any `HaveParts` message `hp` to `n1` that contains `wp` (AD3).

- Description: Node `n1` sends a spurious `WantParts` message.
- Related properties: AL4.
- Inspection results:

During the regular flow of the parts propagation, the `WantParts` is sent in the `requestFromPeer` (code ref↗) function, after invoking it from the method `handleHaves` (code ref↗). Each part from the `HaveParts` message is queued in the `receivedHaves` channel and processed in the `requestFromPeer` function. `sendWantsThenBroadcastHaves` will create and send a `WantParts` message. However, the `WantParts` message sending can fail. See the finding

`HaveParts` broadcast despite the failed `WantParts` send. Additionally, note the finding ***Race condition in `HaveParts` processing during height transitions***.

The property doesn't hold for the catch-up model, as intended. This flow is analyzed within the threat model for the catch-up mechanism.

ALT5: A (non-proposer) node `n1` receives a `WantParts` message `wp` from peer `p`, and `n1` does not drop `wp` because of DV4a, DV4b, or DV4c, and `n1` did not previously send `p` a `HaveParts` message `hp` that contains `wp` (AD3). Node `n1` does not drop `wp`, or does not disconnect from `p`.

- Description: Node `n1` accepts a spurious `WantParts` message.
- Related properties: AL5, AL15.
- Inspection results:

As described in the conclusion of ***DVT8***, the code doesn't implement this check.

ALT6: A (non-proposer) node `n1` receives a `WantParts` message `wp` from peer `p`, and `n1` does not drop `wp` because of DV4a, DV4b, DV4c, or AL5; `cb` is the `CompactBlock` to which `wp` is linked (DD8). Node `n1` does not send any `RecoveryPart` message `rp` linked to `cb` to `p`, and does not add (AD4) `p` to set *outstanding(*`n1`, `wp.height`, `wp.round`, *i)*, for any *i*.

- Description: Node `n1` does not reply to a legitimate request for parts - omission fault.
- Related properties: AL6.
- Inspection results:

The threat does hold. When a valid `WantParts` message is received, the node retrieves the relevant parts (code ref↗). If it possesses any of the requested parts, it iterates through them and sends a `RecoveryPart` message for each (code ref↗). For any parts listed in the `WantParts` message that the node does not have, those parts are added to the missing parts list (code ref↗). However, if the `TrySendEnvelopeShim` fails, the part will never be added to the list of missing parts. See the finding ***`TrySendEnvelopeShim` can silently fail in `handleWants` function***.

However, for the nodes in catch-up mode, see the finding ***When clearing wants the node sends parts without proof***.

ALT7: A (non-proposer) node `n1` sends a `RecoveryPart` message `rp` to a peer `p`. Peer `p` has not sent a `WantParts` message `wp` to `n1` such that `wp` contains `rp` (AD5).

- Description: Node `n1` sends a spurious `RecoveryPart` message.
- Related properties: AL7.
- Inspection results: There is no risk of this threat occurring. `n1` only sends `rp` when handling received `WantParts` messages (code ref↗) and when clearing previously received `WantParts` messages (code ref↗). `n1` only sends recovery parts that it previously received, which means `rp` is linked to a `CompactBlock` `cb` (see property DV2a). Also, `n1` only accepts `wp` if it is linked to a `CompactBlock` `cb` (see property DV4a). The `rps` sent as a result of receiving a `wp` are linked to the same `cb` as the `wp` (code ref↗). Finally, `n1` sends only parts that have the corresponding bit in `wp.parts` set (cod ref↗).

ALT8: A (non-proposer) node `n1` receives a `RecoveryPart` message `rp` from peer `p`, and `n1` does not drop `rp` because of DV2a, DV2b, or AL14, and `n1` did not previously send `p` a `WantParts` message `wp` that contains `rp` (AD5). Node `n1` does not drop `rp`, or does not disconnect from `p`.

- Description: Node `n1` accepts a spurious `RecoveryPart` message.
- Related properties: AL8, AL15.
- Inspection results: When receiving `rp`, `n1` doesn't check that it previously sent a `wp` that contains `rp`. In practice, this is probably fine as `n1` either drops `rp` because of DV2a, DV2b, or AL14, or it handles `rp` as it's valid. Also, as property AL15 doesn't hold, `n1` would anyway not disconnect from `p`. Note the finding ***Nodes don't drop `RecoveryPart` message if they were not requested***.

ALT9: A (non-proposer) node `n1` receives a `CompactBlock` message `cb` from `p1` for the first time, and does not drop it because of DV1. Node `n1` does not send `cb` to one of its peers `p2`, where `p1` $\neq$ `p2`.

- Description: Node `n1` does not properly propagate a `CompactBlock`.
- Related properties: AL9.
- Inspection results: Once a compact block passes validation and the proofs can be added to the cache, the function `broadcastCompactBlock` is called, which broadcasts the block to all peers except the one it was received from. Note that again, there is no retransmission in case of failure. See the finding ***TrySendEnvelopeShim can silently fail in broadcastCompactBlock function***.

ALT10: A (non-proposer) node `n1` receives a `HaveParts` message `hp` for the first time, and does not drop it because of DV3a, or DV3b. Let `hp'` be a `HaveParts` message that contains all parts from `hp` for which `n1` has not sent a `HaveParts` message before. (a)`hp'` is empty but `n1` sends a HavePart message, or (b) `hp'` is non-empty but node `n1` either does not relay `hp'` at all, or (c) sends another `HaveParts` message `hp2 != hp'` to its peers (except the peer that `n1` received `hp` from).

- Description: Node `n1` does not properly propagate a `HaveParts` message.
- Related properties: AL10.
- Inspection results:

The threat holds. See the findings ***TrySendEnvelopeShim can silently fail in broadcastHaves function***, ***Race condition in HaveParts processing during height transitions, Incorrectly sized `maxRequests` BitArray allows unlimited requests for parity parts***, and ***The concurrent request limit might be computed inaccurately***.

ALT11: A (non-proposer) node `n1` receives a `RecoveryPart` message `rp`, and `n1` does not drop `rp` because of DV2a, DV2b, or AL14. Node `n1` does not send `rp` to any peer in the set *outstanding(*`n1`, `rp.height`, `rp.round`, `rp.index`*)*.

- Description: Node `n1` does not properly propagate a `RecoveryPart` message to the peers that had previously requested it.
- Related properties: AL11.
- Inspection results: When the node commits a new block with height `H`, it prunes all the peer states for heights < `H` (code ref↗). This entails clearing the list of wants for the pruned heights `h` (i.e., the `outstanding(n1, h)` set). This could happen before the node has a chance to send all the received recovery parts messages (code ref↗). See the finding ***`ClearWants` might fail due to pruning***.

ALT12: (dropped)

ALT13a: Proposer node `n` is proposing block *b*. Node `n` derives `CompactBlock` instance `cb` from *b* (AD1). The proposer node `n` sends a `HaveParts` message `hp_i` to each of its peers `p_i`. There is a part *p* $\in$ *parts(serialised(b)) U parity(serialised(b))* that is not advertised in any field `hp_i.parts`.

- Description: Proposer node `n` is not correctly load balancing the parts of a block: there is a part `n` is not advertising to any of its peers.
- Related properties: AL12.
- Inspection results:

The threat does not hold. When proposing a block in the `ProposeBlock` method, the `chunkParts` function ensures that all parts are assigned to peers, regardless of the number of peers or parts (code ref↗). Parts are divided into chunks, and each chunk is assigned to one or more peers. Note that the `redundancy` is set to a constant value of 1 (code ref↗).

ALT13b: Proposer node `n` is proposing block *b*. Node `n` derives `CompactBlock` instance `cb` from *b* (AD1). Proposer node `n` does not send a `HaveParts` message `hp_i` to any peer.

- Description: The Proposer node `n` is dropping one of the `HaveParts` messages. Omission fault.

- Related properties: AL12.
- Inspection results:

The threat holds. When `HaveParts` are broadcasted in the `ProposeBlock` function, the `TrySendEnvelopeShim` function may return an error and fail silently, allowing the loop to continue (code ref↗). As a result, some `HaveParts` messages may not be sent. If the send fails for peer `N`, all parts in `chunks[N]` may become unavailable, since that peer could be the only one assigned those parts. See the finding ***Block parts can become unavailable due to silent failure in the TrySendEnvelopeShim function***.

ALT14: Let `m` be an instance of `CompactBlock`, `HaveParts`, `WantParts`, or `RecoveryPart`. Node `n1` sends `m` twice to a peer `p`.

- Description: The analysis covers each of the message types.
- Related properties: AL10, AL13.
- Inspection results:

`CompactBlock`

The threat does not hold. A compact block is sent at two places: Reactor.AddPeer and Reactor.broadcastCompactBlock.

- broadcastCompactBlock is called from handleCompactBlock only, and only if AddProposal returns true. This is the case only once for each height and round. Thus no node sends a compact block twice to a distinct peer by this function.
- AddPeer(peer) is called once a new peer is added, thus the compact block has not been sent before. The function sends a compact block for the current height and round only if it is in the proposal cache, and thus any further invocation of broadcastCompactBlock will not send the compact block again.

`HaveParts`

The threat doesn't hold for the `HaveParts` messages. When sending want messages and broadcasting (code ref↗) haves, a node broadcasts haves for parts it has received in a `HaveParts` message (code ref↗), provided that it did not previously possess those parts (code ref↗). The peer processes a `HaveParts` message and places the request in the `receivedHaves` channel (code ref↗). The `requestFromPeer` function checks whether the part has already been requested from that peer (code ref↗). If it has, the want message is not sent again. As a result, the `sendWantsThenBroadcastHaves` function is not called a second time for the same part from the same peer.

Simultaneously, the `broadcastHaves` function is invoked for parts that the node has recovered locally (code ref↗). Prior to the `HaveParts` message broadcasting in `recoverPartsFromMempool`, the `AddProposal` check in `handleCompactBlock` ensures that only one go routine handles each compact block (code ref↗). This guarantees that the `HaveParts` message is broadcast exactly once per compact block (code ref↗).

Additionally, haves are broadcast only once within the `handleRecoveryPart` function, triggered when the block is successfully decoded for the first time (code ref↗). When a node receives sufficient parts to decode the entire block (code ref↗), it broadcasts to all its peers `HaveParts` messages for all the parts (original and parity). This might lead to some peer asking for all these parts before it manages to decode the entire block, which would result in unnecessary transfer of redundant data. Although this issue cannot affect the correctness of the protocol, it might affect performance, which is the main goal of the propagation reactor. However, since the number of requests a peer can send to the node is bounded, given 100 validators, the peer cannot ask the node for more than `N/34` parts at the same time. In the worst-case scenario, this might lead to a maximum of 3% more data transferred, which is negligible.

In the `ProposeBlock` function, for the proposer, part distribution is performed once per peer. A given chunk may be sent to multiple peers, but each peer receives it only once (code ref↗). Since `handleCompactBlock` is called with the proposer flag set to true, the `recoverPartsFromMempool` function is not triggered for the proposer (code ref↗).

`WantParts`

Similarly, as noted in the `HaveParts` section, the threat doesn't hold for the `WantParts` messages. The peer processes a `HaveParts` message and places the request in the `receivedHaves` channel (code ref↗). The `requestFromPeer` function checks whether the part has already been requested from that peer (code ref↗). If it has, the want message will not be sent again. As a result, the `sendWantsThenBroadcastHaves` function is not called a second time for the same part from the same peer.

In the catch-up mechanism, the tracking of peer requests ensures that duplicate requests won't be sent to the same peer. When parts are requested from a peer, they're recorded via `AddRequests` (code ref↗), and before sending new requests, existing requests are checked through the `GetRequests` function (code ref↗).

`RecoveryPart`

There is no risk of this threat occurring for `RecoveryPart` messages. `n1` sends to `p` a `RecoveryPart` message `rp(h, r, i)` (for height `h`, round `r`, and index `i`) only if it receives from `p` a `WantParts` message `wp` that contains `rp`. If `n1` has `rp` when it receives `wp` from `p`, then it sends it to `p` immediately (code ref↗). Otherwise, `n1` adds `p` to its *outstanding(n1, h, r, i)* set (code ref↗). Once `n1` receives `rp` for the *first* time, it adds it to its own `CombinedPartSet` (code ref↗) and then it sends it to all peers in *outstanding(n1, h, r, i)* (code ref↗) and clears the *outstanding* set (code ref↗). Since `p` doesn't sent `wp` more than once (according to property AL13, for `WantParts`), `n1` cannot send `rp` more than once.

ALT15: Let `m` be an instance of `CompactBlock`, `HaveParts`, `WantParts`, or `RecoveryPart`. Node `n1` receives `m` twice from a malicious peer `p`. Upon receiving `m` the second time, node `n1` does not drop `m`, or does not disconnect from `p`.

- Description: Node `n1` is accepting duplicate messages coming from the same peer.
- Related properties: AL10, AL14, AL15.
- Inspection results:

Property AL15 doesn't hold (i.e., nodes do not disconnect from peers). This issue is documented in the ***The specified disconnection rules were not implemented*** finding and was remarked by the clients as a known vulnerability at the start of the audit.

The following conclusions address whether `n1` is dropping `m` upon receiving it a second time.

`CompactBlock`

When another second valid CompactBlock is received for the same height and round (irrespectively of the sender) AddProposal will not add the new block and no further action takes place as the function returns false. That is, the message will be ignored. There is no disconnection.

`HaveParts`

When a duplicate `HaveParts` message is received, its validity is verified as described in the DVT4 and DVT5 threat descriptions. If the message is valid, the parts listed in the `HaveParts` message are processed by the `requestFromPeer` function. Any part for which a `want` has already been sent to the peer that sent the `HaveParts` message will be ignored (code ref↗).

`WantParts`

The threat holds. There is no explicit protection against receiving duplicate `WantParts` messages (code ref↗). Note the finding ***Nodes don't drop duplicate `WantParts` message.***

`RecoveryPart`

If `m` is invalid, `n1` drops it (see argument for DVT3). Otherwise, the first time `n1` received a `RecoveryPart rp`, it added it to its own `CombinedPartSet` (code ref↗). As a result, when receiving the same `rp` again, it will drop it when trying to add it to the `CombinedPartSet` (code ref↗). Note that if `rp` becomes invalid by the time `n1` receives it for the second time, `n1` will drop it (see argument for DVT3).

ALT16: A node `n1` sends a message `m` that is not valid. Message `m` can be an instance of `CompactBlock` (DD2b), `HaveParts` (DD7), `WantParts` (DD9), or `RecoveryPart` (DD5).

- Description: Node `n1` is sending invalid messages to the network, either because it did not validate them, or because of a bug/malicious behaviour.
- Related properties: AL16
- Inspection results:

`CompactBlock`

A compact block is sent either by the proposer or retransmitted if previously received. The correct sending by the proposer is analyzed in Threat ALT1. A block is stored and retransmitted only if it passed validation after reception. This is analyzed in Threat DVT1. Both threats partially hold. This threat (ALT16) does not add any findings to the previous analysis.

`HaveParts`

The threat holds for `HaveParts` messages. When the `ProposeBlock` function is called, the `HaveParts` message is tightly linked to the associated `CompactBlock`, since the `CompactBlock` is created and stored through the `handleCompactBlock` function (code ref↗) before the `HaveParts` message is sent (code ref↗). The `CompactBlock` is also validated within `handleCompactBlock` (code ref↗). When creating the `HaveParts` message, the proposer uses `part.Proof.LeafHash` values from already validated parts (code ref↗). These hashes match the `PartsHashes` in the `CompactBlock`, ensuring consistency(code ref↗). The indices used in the `HaveParts` message are derived from the block, which includes both original and parity parts (code ref↗). The `BitArray` in this set is sized exactly to match the total number of parts (code ref↗) and is fully populated (code ref↗). This guarantees that when the function `chunkToPartMetaData` is called, any index returned by `chunk.GetTrueIndices()` is valid and corresponds to an actual part (code ref↗). The hashes in the `HaveParts` message match the `cb.parts_hashes` field (code ref↗) since the same parts are used to construct the `PartsHashes` in the `CompactBlock` (code ref↗).

During the `handleCompactBlock` function, the `CompactBlock` is validated before `recoverPartsFromMempool` is called (code ref↗). When constructing the `HaveParts` message, the height and round are taken directly from `cb.Proposal` (code ref↗). Each part included in the `HaveParts` message is constructed using data derived from the `CompactBlock` (code ref↗). The proofs are obtained and validated through `cb.Proofs` (code ref↗). Additionally, the indices used for part metadata in the `HaveParts` message correspond to the ones in the `CompactBlock` (code ref↗).

When broadcasting haves from `sendWantsThenBroadcastHaves`, the haves are constructed using the `convertWantToHave` function (code ref↗). As described in the conclusion of the ALT2 thread, the `HaveParts` are tightly linked to the `CompactBlock`, which is validated when `handleHaves` is invoked (code ref↗). The hashes are then verified through the `ValidatePartHashes` method (code ref↗). The `convertWantToHave` function uses the `getAllState` function to retrieve the appropriate compact block with the correct height and round. However, the set `Index` value (code ref↗) could be inconsistent due to the issue described in the finding ***Race condition in `HaveParts` processing during height transitions.***

As discussed in the ALT2 threat, the `HaveParts` message will not be broadcast within the `handleRecoveryPart` function during catchup mode. Otherwise, the available parts are associated with the compact block retrieved through the `getAllState` function, which is then validated (code ref↗). Each part is accessed using the `parts.GetPart` function (code ref↗), which includes index bounds checking (code ref↗). This function retrieves a part that has already been added and validated, either through `AddPart` with proof validation (code ref↗), or via `recoverPartsFromMempool`, where they are pre-verified using `cb.Proofs()` (code ref↗). Index validity is additionally ensured by checks within the `handleRecoveryPart` function and is bounded by `parts.Total` (code ref↗).

`WantParts`

The threat holds. As described in the conclusion of the ALT2 thread, `SendWantsThenBroadcastHaves` is only called in response to receiving `HaveParts` messages from peers (code ref↗), when the `receivedHaves` channel is populated in the `handleHaves` function (code ref↗). The `handleHaves` function checks the existence of the compact block and validates the part hashes (code ref↗). A node won't send `WantParts` without first receiving and validating

`HaveParts` against a `CompactBlock`. However, a node can request parts that it didn't receive haves for. See the finding ***Race condition in `HaveParts` processing during height transitions***.

The bits are set based on the `have.index` values that come from the `receivedHaves` channel in the `handleHaves` function (code ref↗). At least one bit will be set because `GetTrueIndices` only returns indices where bits are set to true, and the loop in `requestFromPeer` only exits after setting at least one bit (code ref↗).

When `WantParts` is created through `requestFromPeer` (code ref↗), the `parts.Total` comes from the total of original and parity data of `parts CombinedPartSet`, which is retrieved from `getAllState` (code ref↗).

In case of regular flow, the `wp. prove` is set to false since, according to the proto3 semantics, the default value for bool fields is false (code ref↗).

During the catchup, in `retryWants`, the `Parts` field of the `WantParts` message comes from the function `MissingOriginal` (code ref↗), which returns a `BitArray` of size `original.Total() * 2` (code ref↗). This ensures that the total size equals the sum of the original data length and the parity data length. Note that the `CompactBlock.PartsHashes` has size `ParityRatio * total`, where `ParityRatio = 2` (code ref↗). The check `!missing.IsEmpty()` ensures that at least one bit in `wp.parts` is set before sending (code ref↗). Additionally, when constructing the `WantParts` message, `prove` is set to true (code ref↗).

`RecoveryPart`

There is no risk of this threat occurring for `RecoveryPart` messages. `n1` sends to `p` a `RecoveryPart` message `rp(h, r, i)` (for height `h`, round `r`, and index `i`) only if it first added it to its own `CombinedPartSet`, i.e., `n1` constructs `rp` either by retrieving the part from its own `CombinedPartSet` (code ref↗ & ref↗) or by creating a copy of a received `rp` that it successfully added to `CombinedPartSet` (code ref↗).

Furthermore, `n1` adds a new part to its own `CombinedPartSet` in one of the following scenarios:

1. When it proposes a new proposal (code ref↗).
2. When it recovers the part from mempool (code ref↗).
3. When it receives a `RecoveryPart` message from a peer (code ref↗)
4. When it decodes the parts (code ref↗).

In all the cases, the part is only added if it is linked to a `CompactBlock` from the proposal cache (code ref↗ & ref↗ & ref↗).

In the first case, `n1` creates a compact block s.t. `cb.parts_hashes[rp.index] = hash(rp.data)` (code ref↗) and the proofs are correctly constructed during the creation of the original parts (code ref↗) and during encoding (code ref↗).

In the second case, `n1` only recovers the original parts. The proofs are generated using the `parts_hashes` of the linked `cb` (code ref↗). However, `cb.parts_hashes[rp.index] = hash(rp.data)` is not explicitly checked. See the finding ***The parts retrieved from mempool are not being validated***.

In the third case, `n1` only accepts valid `RecoveryPart` messages (see argument for DVT3).

In the fourth case, if the decoding succeeds, then this means the encoding was correct and as a result, `cb.parts_hashes[rp.index] = hash(rp.data)` for all decoded parts. Also, all the proofs are being reconstructed correctly during the decoding (code ref↗).

ALT17: A (non-proposer) node `n1` has run the High Throughput Recovery algorithm, respecting all properties in this threat model. Node `n1` is eventually able to reconstruct the proposed block *b*. Block *b* is invalid (in the sense of CometBFT block validation). Node `n1` accepts *b*.

- Description: If a proposer node `n` is malicious, it may have produced an invalid proposed block. Then, if `n` follows all rules in this model to distribute the block, the only point at which other nodes can realise the block is invalid is when they are able to reconstruct the block and validate it (as CometBFT does).

- Related properties: AL18.

- Inspection results: The node is executing the `syncData` method in a separate go routine (code ref↗). This logic executes the following steps in a loop (every 150ms):
  - It gets the height `h` and round `r` the consensus reactor is working on (code ref↗).
  - It gets from the propagation reactor the proposal and original parts for `h` and `r` (code ref↗).
  - If the consensus reactor doesn't have a proposal yet and the propagation reactor does, it sends it via the `peerMsgQueue` channel (code ref↗).
  - For every part that is missing from the consensus reactor and it's available in the propagation reactor, it sends it via the `peerMsgQueue` channel (code ref↗).

  Once the consensus reactor receives all the original parts, it handles them using logic that was not changed by the code under audit (code ref↗).

  Note the finding ***Race condition in syncData causes a runtime panic***.

Threat model for catchup mechanism

Protocol Invariants

Overall safety property: If a block is committed for a height `h` by a correct process, no process commits on a different block for height `h`.

This is ensured by the properties of consensus and the use of signatures on blocks. There are no changes to this due to the change of proposal block propagation.

Overall liveness property: If a block is committed, every correct process eventually learns about this block.

With the base protocol of block propagation it could be the case that some processes do not receive all needed block parts for a past height `h` while the blockchain already progressed to a larger height `h'`.

The concern is addressed by a catchup protocol, that takes the following steps:

Sub-Properties

From the description of the catchup protocol in system overview section, we can deduce the following sub-properties:

Liveness:

P-C1: A node will keep requesting block parts from new peers until it either received the block part or it requested it from all its peers.

P-C2: If a node `n1` requests a block part in the catchup mechanism from another node `n2`, then `n2` eventually receives this request.

P-C3: If a node receives a block part request (`WantParts`), it responds to this request either immediately if it has this part or later once it receives this part.

P-C4: A block part that is sent is eventually received and stored by the requesting node.

P-C5: If a node received all block parts for a previous height and round it eventually reconstructs the block correctly and commits the block for that height and round.

P-C6: A block created in the catchup mechanism needs to be valid

Performance:

P-C7: No node requests a block part it already has.

Threats

Property P-C1

T-C1: A node `n1` does never request a block part `bp` from a peer `n2`, although the block part remains missing forever.

Conclusion: The threat holds. When a node has a stale `currentHeight` in the propagation reactor state, the missing block parts corresponding to that height are skipped by `retryWants`. See the finding ***Stale `currentHeight` in propagation reactor causes catchup to skip blocks***. In addition, in `retryWants`, the node is incorrectly updating it's outstanding requests which results in the node never requesting some missing block parts as explained in the finding ***The requests made in a step of catchup are incorrectly updated***.

Property P-C2

T-C2: A node `n1` requests a block part from a correct peer `n2`, but `n2` never receives the request.

Conclusion: The threat holds. In `retryWants` , the requests already made to a peer are incorrectly tracked as mentioned in the finding ***The requests made in a step of catchup are incorrectly updated***. This may cause the node to never re-transmit requests to a peer in case `p2p.TrySendEnvelopeShim` fails to deliver the request, given that the failed request can be labeled as delivered.

Property P-C3

T-C3a: A node `n2` that receives a block part request for bp from a peer `n1` and has this block part does not send it to `n2`. Conclusion: The threat holds. As noted in the conclusion of threat ALT6, when a valid `WantParts` message is received, the node retrieves the relevant parts (code ref↗). If it possesses any of the requested parts, it iterates through them and sends a `RecoveryPart` message for each (code ref↗). However, the same issue persists ***TrySendEnvelopeShim` **can silently fail in** `handleWants` **function***.

T-C3b: A node `n2` that receives a block part request for `bp` from a peer `n1` and does not have `bp` but receives `bp` later does not send `bp` to `n1`.

Conclusion: The threat holds. After successfully handling a recovery part, the `clearWants` go routine is spun out to send the `RecoveryPart` to peers that previously requested it. According to the finding ***When clearing wants the node sends parts without proof***, the `prove` field from the original `RecoveryPart` is discarded by `clearWants`, resulting in the original catchup part not being forwarded to the concerned peers. Additionally, sending of the `RecoveryPart` can fail, note the finding ***`TrySendEnvelopeShim` can silently fail in `handleWants` function***.

Property P-C4

T-C4: A node receives a valid block part `bp` that is missing, but does not store it in the proposal cache

Conclusion: The threat holds. As we outlined in the finding ***AddCommitment doesn't update the PartSetHeader for cached heights and rounds***, if a node receives a valid block part for which the cached `CombinedPartSet` corresponding metadata is stale, the call to `AddParts` in `handleRecoveryPart` (code ref↗) will fail resulting in the valid part being discarded.

Property P-C5

T-C5a: A node has all parts of a block in its proposal cache but never reconstructs the block.

Conclusion: There is no risk for this threat to occur. A node that has all parts of a block in its proposal cache will always trigger reconstruction because the consensus reactor continuously extracts complete proposals and missing block parts from the consensus reactor via the `syncData` routine (code ref↗).

`syncData` periodically checks whether the consensus reactor already has a complete proposal and its block parts(code ref↗). If the current proposal is not found or not complete yet, It obtains the current height and round, then calls `GetProposal` (code ref↗) on the propagation reactor.

If the proposal is found in the propagation reactor and has a valid signature (meaning that it was added to the propagator via a `CompactBlock` in the happy path), `syncData` sends an internal `ProposalMessage` into `cs.peerMsgQueue` to copy the proposal from the state of the propagation reactor into the state of the consensus reactor(code ref↗). If the proposal returned by the propagator has no `Signature` field, it was introduced by the catch-up mechanism via `AddCommitment` (code ref↗). By design, the consensus reactor has prior knowledge of such proposal given that `AddCommitment` is triggered only by the consensus reactor.

Then, for every missing block part that exists in the propagation reactor but doesn't exists in the consensus reactor (code ref↗), `syncData` dispatches an internal `BlockPartMessage` for that part (code ref)↗.

The handling of the message receipt from the concerned message queues is assumed to be correct per the correctness guarantees of `CometBFT`.

T-C5b: The proposal cache gets pruned for height `h` before h was committed.

Conclusion: There is no risk for this threat to occur. The pruning of the state in the propagation reactor has a single entry point `Prune` (code ref↗) which is triggered only by the consensus reactor upon calling `finalizeCommit` (code ref↗). `finalizeCommit` on height `h` will cause the propagation reactor to delete the cached proposals for all heights lower than h, which are guaranteed to be finalized per the correctness guarantees of `CometBFT`.

Property P-C6

T-C6: A node does not check the validity of a block or does not properly drop an invalid block.

Conclusion: The threat holds. As explained in the finding ***AddCommitment doesn't update the PartSetHeader for cached heights and rounds***, if a node receives an invalid block part for which the cached `CombinedPartSet` is outdated, the `handleRecoveryPart` can accept the outdated recovery part (code ref↗).

Property P-C7

T-C7: A node requests a block part it already has.

Conclusion: The threat holds. In `retryWants`, the missing block parts are fetched from the state before the start of the retrial loop (code ref↗). In addition, the requests already made to a peer are incorrectly tracked as mentioned in the finding ***The requests made in a step of catchup are incorrectly updated***. This allows requests for the same part to be made during different steps for different peers. In the event a `RecoveryPart` is received before all the catchup steps are done, a request for that received part can be emitted.

Findings

Finding	Type	Severity	Status
Stale currentHeight in the propagation reactor causes catchup to skip blocks	Protocol	Critical	Patched Without Reaudit
The CompactBlock validation doesn't check whether the proposal is for the right height and round	Implementation	Critical	Resolved
AddCommitment doesn't update the PartSetHeader for cached heights and rounds	Implementation	Critical	Patched Without Reaudit
The requests made in a step of catchup are incorrectly updated	Implementation	High	Patched Without Reaudit
Not validating last part length of CompactBlock leads to a panic while decoding	Implementation	High	Patched Without Reaudit
Calling SetHave and SetWant method could trigger a panic	Implementation	High	Patched Without Reaudit
Block parts can become unavailable due to silent failure in the TrySendEnvelopeShim function	Implementation	Medium	Patched Without Reaudit
Race condition in HaveParts processing during height transitions	Implementation	Medium	Patched Without Reaudit
Race condition in syncData causes a runtime panic	Implementation	Medium	Patched Without Reaudit
TrySendEnvelopeShim can silently fail in handleWants function	Implementation	Medium	Patched Without Reaudit
When clearing wants the node sends parts without proof	Implementation	Medium	Patched Without Reaudit

Finding	Type	Severity	Status
Unresolvable wants in PeerState	Implementation	Medium	Patched Without Reaudit
The parts retrieved from mempool are not being validated	Implementation	Medium	Patched Without Reaudit
if !blockProp.started.Load() check missing from the handleCompactBock function	Implementation	Low	Disputed
The specified disconnection rules were not implemented	Protocol	Low	Risk Accepted
Propagation reactor adds peers without verifying PBBT support	Implementation	Low	Disputed
Incorrectly sized maxRequests BitArray allows unlimited requests for parity parts	Implementation	Low	Patched Without Reaudit
ClearWants might fail due to pruning	Protocol	Low	Acknowledged
The concurrent request limit might be computed inaccurately	Implementation	Low	Acknowledged
The check for complete CombinedPartSet during catchup is incorrect	Implementation	Low	Patched Without Reaudit
Catchup on cached proposals is delayed until next reactor tick	Implementation	Low	Patched Without Reaudit
Proofs are verified for received parts that the node already has	Implementation	Low	Patched Without Reaudit
HaveParts broadcast despite the failed WantParts send	Implementation	Low	Acknowledged
TrySendEnvelopeShim can silently fail in broadcastCompactBlock function	Implementation	Low	Acknowledged
TrySendEnvelopeShim can silently fail in broadcastHaves function	Implementation	Low	Acknowledged

Finding	Type	Severity	Status
GetPart returning nil causes a runtime panic	Implementation	Informational	Acknowledged
Discrepancy between the implementation and specification	Documentation	Informational	Risk Accepted
TxMetaData cannot be validated before block is complete	Protocol	Informational	Risk Accepted
Nodes don't drop duplicate WantParts message	Protocol	Informational	Acknowledged
Nodes don't drop RecoveryPart message if they were not requested	Protocol	Informational	Acknowledged
Miscellaneous code findings	Implementation	Informational	Acknowledged

Stale currentHeight in the propagation reactor causes catchup to skip blocks

Severity Critical **Impact** 3 - High **Exploitability** 3 - High

Type Protocol **Status** Patched Without Reaudit

Involved artifacts

- `consensus/propagation/catchup.go`↗
- `consensus/propagation/commitment_state.go`↗
- `consensus/propagation/have_wants.go`↗

Description

Whenever `AddCommitment` (code ref↗) is invoked for a proposal at height *h* and round r, the corresponding proposal is inserted into the propagation reactor's proposal cache without updating the cache's `currentHeight` (or `currentRound`) to reflect this newly committed proposal. On the other hand, the catchup logic in `retryWants` iterates over every `height` ≠ `currentHeight`, issuing requests for the missing block parts (code ref↗).

This results in the catchup mechanism skipping the block height corresponding to the stale `currentHeight` (a height from which the rest of the rest of network has passed, given that more recent heights were committed). In addition, the outstanding requests for the block at `currentHeight` won't get answered by the rest of the network that is propagating a new block with a more recent height, given that the `WantParts` messages will fail the relevancy checks in `handleWants` (code ref↗).

Problem scenarios

Consider a scenario where a subset of nodes is slow and their `currentHeights` are behind the rest of the network as they can't finalize blocks without fully receiving them. The majority of the network is faster and can form quorums without requiring the slow nodes, so the consensus is advancing, and slow nodes are seeing the blocks being committed, triggering `AddCommitment` for each new height. When the catch-up loop runs, it skips the unfinished block at `currentHeight`. The block at height `currentHeight` may never be finalized on the slow nodes. In addition, the affected nodes will be stuck running the catchup mechanism for all subsequent block heights, which consumes significantly more resources than the optimistic pull-based propagation for the whole network.

Recommendation

We recommend revising the update procedure for the `currentHeight`, `currentRound`, `consensusHeight`, `consensusRound` fields in the `ProposalCache` (code ref↗) to take into account that new heights can be introduced via paths other than `AddProposal` (code ref↗).

Status

Patched without re-audit↗.

Based on the initial review, we have not conclusively verified that the root issue cause has been resolved and we recommend further review to confirm.

The CompactBlock validation doesn't check whether the proposal is for the right height and round

Severity Critical **Impact** 3 - High **Exploitability** 3 - High

Type Implementation **Status** Resolved

Involved artifacts

- `consensus/propagation/commitment.go`↗
- `consensus/state.go`↗
- `consensus/propagation/commitment_state.go`↗

Description

When handling the incoming compact block, the function `validateCompactBlock` is called (code ref↗). This function calls the `ValidateProposal` method, which validates the `cb.Proposal` (code ref↗). This function doesn't validate whether the proposal's height and round match the expected consensus state (code ref↗).

Note that in the current implementation, the `validateCompactBlock` function is not complete (code ref↗).

Problem scenarios

The function `AddProposal` ensures that for the incoming compact block, the proposal height is less than `consensusHeight`, and that the round is less than `consensusRound` (code ref↗). However, consider a scenario where a malicious proposer proposes a block for the height max(int64). Eventually, this proposal will timeout as it will be rejected by the consensus reactor, but it will be set be the receiving nodes as the current height (code ref↗).

Recommendation

We recommend adding the following check:

```
if proposal.Height != cs.Height || proposal.Round != cs.Round {
    return nil
}
```

Status

Resolved↗.

AddCommitment doesn't update the PartSetHeader for cached heights and rounds

Severity Critical **Impact** 3 - High **Exploitability** 3 - High

Type Implementation **Status** Patched Without Reaudit

Involved artifacts

- `consensus/state.go`↗
- `consensus/propagation/catchup.go`↗

Description

The consensus reactor calls `AddCommitment` each time it realizes it's assembling parts for an incorrect proposal and must reset `cs.ProposalBlock` and `cs.ProposalBlockParts` in the consensus state (code ref1↗, code ref2↗, code ref3↗). When the current proposal is reset, the node updates the block-parts metadata held in the `psh` `PartSetHeader` (code ref↗) and then passes this updated header to `AddCommitment`. However, in `AddCommitment`, once the updated `psh` is used to build a new `CombinedPartSet` (code ref↗), if an earlier proposal was introduced beforehand for that height and round via the happy path as compact block, the updated `psh` never gets persisted to the proposal cache (code ref↗) and the block parts already stored received are not purged. This can cause the node to propagate and request block parts using outdated metadata, potentially leading to the reconstruction of incorrect blocks.

Problem scenarios

Consider a scenario where a malicious leader targets a node n with a compact block while broadcasting a different compact block to the rest of the network. Once a quorum of nodes votes for the correct proposal, node n receives those votes, causing its consensus reactor to reset the current proposal and invoke `AddCommitment`. But because the propagation reactor doesn't save the refreshed `psh` in the proposal cache, node n reverts to the catch-up loop with outdated metadata. The attacker then provides corrupted block parts that still align with the stale header, leading node n to reassemble an invalid block.

Recommendation

We recommend extending `AddCommitment`, purge the cache of the propagation reactor of any parts and metadata tied to the stale proposal in case it is called with a `psh` that doesn't match the cached value.

Status

Patched without re-audit↗.

The requests made in a step of catchup are incorrectly updated

Severity High **Impact** 3 - High **Exploitability** 2 - Medium
Type Implementation **Status** Patched Without Reaudit

Involved artifacts

- `consensus/propagation/catchup.go`↗

Description

During the catchup mechanism, in the `retryWants` method, the requests made to a peer is updated by adding the missing requests instead of the ones sent to the peer (code ref↗). This might lead to nodes not receiving the entire data.

Problem scenarios

This issue occurs every time the `retryWants` method is called.

Recommendation

Our recommendation is to replace `missing` with `mc` in the call to `AddRequests` (code ref↗).

Status

Patched without re-audit↗.

Not validating last part length of CompactBlock leads to a panic while decoding

Severity High **Impact** 3 - High **Exploitability** 2 - Medium

Type Implementation **Status** Patched Without Reaudit

Involved artifacts

- `consensus/propagation/commitment.go`↗

Description

When a compact block is received, in `validateCompactBlock` (code ref↗), there is no check on the validity of `last_len`.

Problem scenarios

When a compact block is received (code ref↗) is not checked if the `last_len` field is smaller than `BlockPartSizeBytes`. Then this value is just copied in the new `CombinedPartSet` (code ref↗). When the block is decoded, the slice is accessed at `[:lastPartLen]` and a runtime panic might occur (code ref↗).

Recommendation

We recommend adding a check in `validateCompactBlock` that validates `last_len <= BlockPartSizeBytes`.

Status

Patched without re-audit↗.

Calling SetHave and SetWant method could trigger a panic

Severity High **Impact** 3 - High **Exploitability** 2 - Medium

Type Implementation **Status** Patched Without Reaudit

Involved artifacts

- `consensus/propagation/peer_state.go`↗
- `consensus/propagation/have_wants.go`↗

Description

Neither `SetHave` nor `SetWant` methods initialize the peer state, which means that if they are called on uninitialized peer state data, it will trigger a runtime panic. Both are called in the `clearWants` method, which is executed in a separate go routine. As a result, if the peer state is pruned (code ref↗) in between calling `WantsPart` (code ref↗) and `SetHave` & `SetWant` (code ref↗), then the code will panic.

Problem scenarios

This issue can occur in the following scenario:

- A node `n1` receives a proposal for height `H` and it recovers all the original parts from mempool (code ref↗).
- `n1`'s consensus reactor finalizes `H` and as a result, `n1` sets `consensusHeight` to `H` (code ref↗).
- `n1` receives a new proposal for height `H+1` and it sets the `currentHeight` to `H+1` (code ref↗).
- `n1` receives one of the parity parts for height `H` and accepts it as neither the `relevant` check (code ref↗) nor the `IsComplete` check (code ref↗) will fail.
- `n1` starts a go routine to clear the wants for the parity part it just received (code ref↗).
- the go routine starts and passes the `WantsPart` check (code ref↗).
- `n1` receives all the necessary parts for height `H+1`, decodes the block, its consensus reactor finalizes the block, and as a result, `n1` prunes all the peer data for height `H` (code ref↗).
- the go routine goes ahead and calls the `SetHave` and `SetWant` methods, which will cause a panic (code ref↗).

Recommendation

Our recommendation is to run the `clearWants` method in the same thread as the one pruning the peer state.

Status

Patched without re-audit↗.

Block parts can become unavailable due to silent failure in the TrySendEnvelopeShim function

Severity Medium **Impact** 3 - High **Exploitability** 1 - Low

Type Implementation **Status** Patched Without Reaudit

Involved artifacts

- `consensus/propagation/commitment.go`↗

Description

When proposing a block in the `ProposeBlock` method, the `chunkParts` function ensures that all parts are assigned to peers, regardless of the number of peers or parts (code ref↗). Parts are divided into chunks, and each chunk is assigned to one or more peers. Note that the `redundancy` is set to a constant value of 1 (code ref↗).

When `HaveParts` are broadcasted, the `TrySendEnvelopeShim` function may return an error and fail silently, allowing the loop to continue (code ref↗). As a result, some `HaveParts` messages may not be sent.

Problem scenarios

If the send fails for peer `N`, due to connection or channel issues, all parts in `chunks[N]` may become unavailable, since that peer could be the only one assigned those parts.

Recommendation

We recommend replacing silent failures with proper error handling and implementing the retry mechanism, as mentioned in the comment (ref↗).

Notably, the clients were aware of the implications of using the `TrySendEnvelopeShim` function prior to the start of the audit.

Status

Patched without re-audit↗.

Race condition in HaveParts processing during height transitions

Severity Medium **Impact** 3 - High **Exploitability** 1 - Low

Type Implementation **Status** Patched Without Reaudit

Involved artifacts

- `consensus/propagation/have_wants.go`↗
- `consensus/propagation/commitment.go`↗
- `consensus/state.go`↗

Description

When a `HaveParts` message is received for `currentHeight` *h+1*, while `consensusHeight` is *h*, it triggers a signal to send requests, and the missing parts are pushed into the `receivedHaves` channel (code ref↗). These are then received by the `requestFromPeer` function through the `ps.receivedHaves` channel (code ref↗). If there are `canSend` available request slots, the logic loops `canSend` times, attempting to retrieve have parts from the `receivedHaves` channel (code ref↗).

Problem scenarios

Consider a scenario where the number of missing parts for height *h+1* is less than `canSend`. In that case, the full `canSend` capacity isn't required.

When the block is finalized, the state is pruned and the `consensusHeight` increases (code ref↗), which allows compact blocks from round *h+2* to be accepted, since they will be able to pass validation (code ref↗).

If during the last part processing for the `currentHeight` *h+1*, after this check↗, the node finalizes the block for that height and receives the compact block and `HaveParts` message for the `currentHeight` *h+2*, the missing parts for the *h+2* height would be added to the `receivedHaves` channel. Since `HaveParts` messages may refer to parts that differ from those retrieved using `getAllState`(code ref↗), this could lead to inaccurate tracking of the request limit (code ref↗) and since different heights may correspond to different numbers of parts, incorrect sizing of the `wantParts` (code ref↗) and requesting parts that we didn't receive haves for because the skipped wants for previous height will be part of the new want.

Recommendation

We recommend updating the `canSend` value to be the minimum of `canSend` and the length of `receivedHaves`, i.e., `min(canSend, len(receivedHaves))`, as well as ensuring that the current height and round of have and wants are the same to avoid race condition.

Status

Patched without re-audit↗.

Race condition in syncData causes a runtime panic

Severity Medium **Impact** 3 - High **Exploitability** 1 - Low

Type Implementation **Status** Patched Without Reaudit

Involved artifacts

- `consensus/state.go`↗
- `consensus/propagation/commitment_state.go`↗

Description

In `syncData`, the consensus reactor fetches the required proposals from the propagation reactor via the `GetProposal` method (code ref↗). `GetProposal` calls `getAllState(height)` which returns a `nil` compact block when the requested `height` is outdated. If that is the case, `GetProposal` would panic (code ref↗).

Problem scenarios

The height passed to `GetProposal` comes from `cs.Height` (code ref↗). However, there's a potential race condition: if there's a delay in the following section (code ref↗), and the propagation reactor advances the height during that time, the compact block returned by `getAllState(height)` will be `nil` and panic will occur.

Recommendation

We recommend adding extra checks in `GetProposal` to avoid de-referencing a `nil` pointer.

Status

Patched without re-audit↗.

The clients noted that this mechanism still needs improvements to handle catchup, and there is an issue↗ and a few proposals on how to fix this.

TrySendEnvelopeShim can silently fail in handleWants function

Severity Medium **Impact** 3 - High **Exploitability** 1 - Low

Type Implementation **Status** Patched Without Reaudit

Involved artifacts

- `consensus/propagation/have_wants.go`↗

Description

In the `handleWants` function, the `canSend` set contains all parts that we could potentially send. If sending one of these parts fails, the loop simply continues without removing the failed part from `canSend` (code ref↗). As a result, when we later calculate `stillMissing` using `wants.Parts.Sub(canSend)`, we subtract all parts in `canSend`, including those that failed to send (code ref↗).

Problem Scenarios

If the `TrySendEnvelopeShim` fails, the part being sent in `RecoveryPart` message won't be included in `stillMissing`. Consequently, these parts are never retried since they weren't successfully sent, nor were they added to `stillMissing`.

A node in catch-up mode may fail to retrieve a part if its only known source doesn't respond. Since each part is requested from a peer only once, and we don't retry with the same peer (code ref↗), a missed response from the sole source makes that part unrecoverable.

Recommendation

We recommend replacing silent failures with proper error handling and implementing the retry mechanism.

Notably, the clients were aware of the implications of using the `TrySendEnvelopeShim` function prior to the start of the audit.

Status

Patched without re-audit↗.

When clearing wants the node sends parts without proof

Severity Medium **Impact** 2 - Medium **Exploitability** 2 - Medium

Type Implementation **Status** Patched Without Reaudit

Involved artifacts

- `consensus/propagation/have_wants.go`↗

Description

When clearing pending `WantParts` messages (i.e., peers requested parts that the node didn't have yet), in the `clearWants` method, the node is sending parts without the proofs (code ref↗). As a result, this will affect the catchup mechanism where peers ask for parts with proofs. When the peers are receiving these parts without proofs, they will just drop them (code ref↗).

Problem scenarios

This issue occurs every time the `clearWants` method is called. However, it requires both the node and the peer to be in the catchup process.

Recommendation

Our recommendation is to forward the entire `RecoveryPart` message, including the proof field.

Status

Patched without re-audit↗.

Unresolvable wants in PeerState

Severity Medium **Impact** 1 - Low **Exploitability** 3 - High

Type Implementation **Status** Patched Without Reaudit

Involved artifacts

- `libs/bits/bit_array.go`↗
- `consensus/propagation/have_wants.go`↗

Description

When handling the incoming `WantParts` message in `handleWants`, the function doesn't check the length of the `wants` (code ref↗). The parts retrieved through the function `getAllState` can be of different sizes from an incoming message (code ref↗). The `ValidateBasic` only ensures that the wants aren't nil (code ref↗).

Only the parts that the node has stored will be sent through the `RecoveryPart` message. The `And` operation handles `BitArrays` of different sizes by operating only on the overlapping bits (code ref↗). This function truncates to the shorter length, and any parts beyond the length of `wants.Parts` are ignored (code ref↗). However, when adding missing parts to the peer's wants (code ref↗), the `Sub` function only iterates up to the minimum length of the two arrays. If the `wants` array is longer, the result will still match its length (code ref↗).

Problem Scenarios

Consider a scenario where a malicious peer sends an invalid `WantParts` message containing a large number of entries in `wants.Parts`. The peer's state ends up containing unresolvable wants, as the requested parts, which are not present in local storage, are added to the list of missing parts. These wants are not cleared until the peer's state is pruned (code ref↗) because non-existent parts were incorrectly recorded as the peer's wants.

Recommendation

We recommend adding a check to ensure that the length of the `wants.Parts` is within acceptable bounds.

Status

Patched without re-audit↗.

The parts retrieved from mempool are not being validated

Severity Medium **Impact** 1 - Low **Exploitability** 3 - High

Type Implementation **Status** Patched Without Reaudit

Involved artifacts

- `consensus/propagation/commitment.go`↗

Description

When recovering parts from the mempool (code ref↗), the node uses the tx metadata provided in the compact block (i.e., a list of tx hashes with start and end positions). This enables the node to reconstruct parts if it has all the encompassing transactions in the mempool. However, the node never checks afterwards if the hash of the reconstructed parts is matching the `parts_hashes` provided in the compact block, i.e., the `cb.parts_hashes[rp.index] = hash(rp.data)` condition from the threat model (DD5).

As a result, a malicious proposer could send a compact block with `parts_hashes` inconsistent with the `blobs` (i.e., the tx metadata). Since `blobs` is only used in recovery from mempool, those nodes that already have the transactions in `blobs` might recovered incorrect parts (i.e., inconsistent with `parts_hashes`). These parts are added without verifying the merkle proofs (code ref↗). Note the the proofs are generated by the node using `parts_hashes`. As a consequence, the node will send invalid `RecoveryParts` messages to its peers. The peers will drop these messages when trying to add the parts and verify the proof (code ref↗).

Once nodes drop the peers from which they receive invalid messages (i.e., once the FMO is implemented), a malicious proposer will be able to get honest peers to disconnect from each other.

Problem scenarios

This issue can occur every time a proposer sends a compact block with inconsistent transaction metadata (i.e., `blobs`).

Recommendation

Our recommendation is to check that the hash of the parts recovered from mempool, match the part hashes in the compact block.

Status

Patched without re-audit↗.

if !blockProp.started.Load() check missing from the handleCompactBock function

Severity Low **Impact** 2 - Medium **Exploitability** 1 - Low

Type Implementation **Status** Disputed

Involved artifacts

- `consensus/propagation/commitment.go`↗

Description

The validation check for if the block propagation reactor has been started is missing in the function `handleCompactBock` (code ref↗).

Problem Scenarios

The reactor could start processing incoming `CompactBlock` messages before its internal state is properly initialized which can cause issues when interacting with the reactor. A proposer could potentially get into an inconsistent state by operating before being properly started.

Recommendation

We recommend adding the following check to the `handleCompactBock` function:

```
if !blockProp.started.Load() {
    return
}
```

Status

Disputed.

The clients noted that they validate proposals before adding them, stating: "We validate proposals before adding them. So if a proposal is valid, then it's fine to add it to the propagation reactor state."

The specified disconnection rules were not implemented

Severity Low **Impact** 1 - Low **Exploitability** 2 - Medium
Type Protocol **Status** Risk Accepted

Involved artifacts

- `consensus/propagation/have_wants.go`↗

Description

The documentation notes that (ref↗):

- *Nodes MUST disconnect from peers that send invalid messages*
- *Nodes MUST disconnect from peers that send `Data` messages that have not been requested*
- *Nodes MUST disconnect from peers that send more than one of the same `Have` or `Want` message*

This is consistent with properties AL14 and AL15 in the threat model.

In the current implementation, the node doesn't disconnect from the peer that send the message in these cases, and consequently, the properties AL14 and AL15 don't hold.

Previously to the audit, the clients noted that the code is relying on peer-to-peer bandwidth constraints:

"*The spec envisions an ideal protocol without per-peer bandwidth restrictions. This can exist, but requires all edge cases to be covered before disconnecting from peers can be added. Since the current protocol does not handle these cases safely, we must retain per-peer bandwidth restrictions to maintain security.*"

Problem scenarios

The issue can occur when a node receives an instance of `CompactBlock`, `HaveParts`, `WantParts`, or `RecoveryPart` message.

Recommendation

Our recommendation is to update the spec to be consistent with the implementation until the planned changes are applied.

Propagation reactor adds peers without verifying PBBT support

Severity Low **Impact** 1 - Low **Exploitability** 2 - Medium
Type Implementation **Status** Disputed

Involved artifacts

- `consensus/propagation/reactor.go`↗

Description

The propagation reactor doesn't check if a peer supports PBBT propagation before adding them (code ref↗) to the `peerstate` map in reactor (code ref↗). This can waste resources by reserving unnecessary storage for unsupported peers and incurring extra iterations in every broadcast or any operation that loops over all peers.

Problem scenarios

This issue can occur when some nodes support the new PBBT propagation mechanism while others still rely on the legacy propagation mechanism.

Recommendation

Before adding a peer to the `peerstate`, the node should check if the peer supports PBBT propagation.

Status

Disputed.

The clients noted that when adding a new peer, during the handshake, the switch sends the list of supported channels including the propagation reactor channels. So, if the handshake passes, then that peer supports PBBT.

Incorrectly sized maxRequests BitArray allows unlimited requests for parity parts

Severity Low **Impact** 1 - Low **Exploitability** 2 - Medium

Type Implementation **Status** Patched Without Reaudit

Involved artifacts

- `consensus/propagation/commitment_state.go`↗

Description

When adding a new `proposalData` (in the `AddProposal` method), the size of the `maxRequests` BitArray is set to the number of the original parts, instead of the total number of combined parts, original plus parity (code ref↗). As a result, the mechanism for limiting the amount of requests per part will not work to parity parts as the check in `requestFromPeer` will always return false for indexes larger or equal than the number of original parts (code ref↗).

Problem Scenarios

The issue will occur every time one node receives more `HaveParts` messages for the same part than the value returned by `ReqLimit` (code ref↗).

Recommendation

We recommend setting the size of the `maxRequests` BitArray to the size of the `totalMap` of the newly created `CombinedPartSet`.

Status

Patched without re-audit↗.

ClearWants might fail due to pruning

Severity Low **Impact** 1 - Low **Exploitability** 2 - Medium

Type Protocol **Status** Acknowledged

Involved artifacts

- `consensus/propagation/have_wants.go`↗
- `consensus/propagation/reactor.go`↗

Description

When the node commits a new block with height `H`, it prunes all the peer state for heights < `H` (code ref↗). This entails clearing the list of wants for the pruned heights `h` (i.e., the `outstanding(n1, h)` set in the threat model). This could happen before the node has a chance to send all the received recovery parts messages (code ref↗). We believe this doesn't affect correctness as the nodes that can no longer get the data via the normal propagation algorithm will use the catchup mechanism. However, if the catchup mechanism is not faster than normal propagation, then these lagging nodes will continue remaining behind more and more.

Problem scenarios

This issue can occur in the following scenario:

- A node `n1` receives a proposal for height `H` and it recovers all the original parts from mempool (code ref↗).
- `n1`'s consensus reactor finalizes `H` and as a result, `n1` sets `consensusHeight` to `H` (code ref↗).
- `n1` receives a new proposal for height `H+1` and it sets the `currentHeight` to `H+1` (code ref↗).
- `n1` receives one of the parity parts for height `H` and accepts it as neither the `relevant` check (code ref↗) nor the `IsComplete` check (code ref↗) will fail.
- `n1` starts a go routine to clear the wants for the parity part it just received (code ref↗).
- `n1` receives all the necessary parts for height `H+1`, decodes the block, its consensus reactor finalizes the block, and as a result, `n1` prunes all the peer data for height `H` (code ref↗).
- the go routine cannot clear wants as it has no peer data for height H anymore (code ref↗).

Note though that since this is a parity part, there is not real impact if the peers of `n1` will not receive it. For this issue to occur for an original part, `n1` must finalize two blocks before the go routine clearing wants gets schedule, which is more unlikely.

Recommendation

Our recommendation is to run the `clearWants` method in the same thread as the one pruning the peer state.

The concurrent request limit might be computed inaccurately

Severity Low **Impact** 1 - Low **Exploitability** 2 - Medium

Type Implementation **Status** Acknowledged

Involved artifacts

- `consensus/propagation/commitment_state.go`↗
- `consensus/propagation/have_wants.go`↗

Description

When computing the concurrent request limit per peer, the number of total parts is taken from the current proposal, the one at `currentHeight` and `currentRound` (code ref↗). However, this limit might be used to bound the number of `WantParts` messages sent for different heights and rounds (code ref↗). Note that the accepted heights and rounds is based on the `consensusHeight` and `consensusRound`.

Problem scenarios

This issue can occur in the following scenario.

- A node `n1` receives a `HaveParts` message for height `H+1`, with `H` the `consensusHeight` (i.e., the height of the latest finalized block). `n1` accepts the message and sends a request for the missing parts to the `receivedHaves` channel (code ref↗).
- `n1` receives the latest missing part for height `H+1`, decodes the block, its consensus reactor finalizes the block, and as a result, `n1` set the `consensusHeight` to `H+1` (code ref↗).
- `n1` receives a new valid proposal for height `H+2` and sets the `currentHeight` to `H+2` (code ref↗).
- the go routine executing the `requestFromPeer` method, calls the `GetCurrentProposal` method and gets the proposal at height `H+2` (code ref↗), and uses the number of parts in this proposal to compute the concurrent request limit (code ref↗).
- the go routine gets the first request from the `receivedHaves` channel, which is the one sent by `n1` for height `H+1` (code ref↗).

Recommendation

Our recommendation is to move the `concurrentReqs` from `PeerState` to `partState` (code ref↗) and to use the total number of parts for the corresponding height and round to compute the concurrent request limit per peer.

The check for complete CombinedPartSet during catchup is incorrect

Severity Low **Impact** 1 - Low **Exploitability** 2 - Medium

Type Implementation **Status** Patched Without Reaudit

Involved artifacts

- `consensus/propagation/types/combined_partset.go`↗

Description

A `CombinedPartSet` is considered to be complete if both original and parity parts are complete (code ref↗). During the catchup mechanism, the node only requests original parts (code ref↗). Furthermore, if the `proposalData` was created via the `AddCommitment` method, then the `catchup` flag in `CombinedPartSet` is set to true (code ref↗), which means the node doesn't attempt to decode any parity parts (code ref↗). This has two consequences. First, in the `retryWants` method, an error log might be hit (code ref↗). Second, the node accepts `RecoveryParts` messages even for blocks that were finalized (i.e., the `relevant` check passes, code ref↗) and only drops them once it tries to add them to the `CombinedPartSet` (after verifying the proof).

Problem scenarios

This issue occurs for every block that is recovered through the catchup mechanism, especially for blocks with `CombinedPartSet` created via the `AddCommitment` method.

Recommendation

Our recommendation is to consider a `CombinedPartSet` complete if the original `PartSet` is complete.

Status

Patched without re-audit↗.

Catchup on cached proposals is delayed until next reactor tick

Severity Low **Impact** 1 - Low **Exploitability** 2 - Medium

Type Implementation **Status** Patched Without Reaudit

Involved artifacts

- `consensus/propagation/catchup.go`↗

Description

In the `AddCommitment` function, it is first checked whether a proposal for the given `height` and `round` is already present in the propagation reactor's `blockProp.proposals` map (code ref↗). If it is, the function returns immediately and does not launch the `retryWants` goroutine (code ref↗). Consequently, the reactor will defer any retries for that proposal until the next reactor tick, up to 6 seconds later. This introduces unnecessary catch-up latency whenever commitments arrive for proposals the reactor has already cached.

Problem scenarios

If a node receives a valid compact block for height h but hasn't fetched all its block parts by the time the network commits that proposal, the ensuing `AddCommitment` sees the cached block entry and returns immediately. Because `retryWants` is only scheduled on the next reactor tick, the node delays re-requesting its missing parts, slowing block finalization.

Recommendation

We recommend starting the `retryWants` goroutine as soon as a commitment is added.

Status

Patched without re-audit↗.

Proofs are verified for received parts that the node already has

Severity Low **Impact** 1 - Low **Exploitability** 1 - Low

Type Implementation **Status** Patched Without Reaudit

Involved artifacts

- `consensus/propagation/have_wants.go`↗
- `consensus/propagation/types/combined_partset.go`↗

Description

When a node receives a `RecoveryPart` message for a part that it already has, it first verifies the proof (code ref↗) and then drops it (code ref↗). Given that the node doesn't disconnect from the peer that sent the message, this can be a spam attack vector.

Problem scenarios

This issue occurs every time a node receives a `RecoveryPart` message for a part that it already has for a block that is not complete yet.

Recommendation

Our recommendation is to exit earlier if the node already has the part.

Status

Patched without re-audit↗.

HaveParts broadcast despite the failed WantParts send

Severity Low **Impact** 1 - Low **Exploitability** 1 - Low

Type Implementation **Status** Acknowledged

Involved artifacts

- `consensus/propagation/have_wants.go`↗

Description

In the `sendWantsThenBroadcastHaves` function, the `sendWant` method is called before `broadcastHaves` (code ref↗). The `sendWant` function returns without an error after `TrySendEnvelopeShim` returns false (code ref↗). If the sending `WantPart` fails on `WantChannel`, it doesn't affect `DataChannel`'s ability to send `HaveParts`. This causes `HaveParts` to be broadcast even though the node failed to request the parts (code ref↗).

Problem Scenarios

The issue can occur in normal operation if the `WantChannel`'s send queue is full. This can lead to inconsistency where nodes advertise parts they haven't successfully requested, potentially causing other nodes to expect parts from peers that aren't actually trying to obtain them.

Recommendation

We recommend only broadcast `HaveParts` if the `WantPart` function succeeds.

TrySendEnvelopeShim can silently fail in broadcastCompactBlock function

Severity Low **Impact** 1 - Low **Exploitability** 1 - Low

Type Implementation **Status** Acknowledged

Involved artifacts

- `consensus/propagation/commitment.go`↗

Description

The `broadcastCompactBlock` function doesn't handle `TrySendEnvelopeShim` failure (code ref↗).

Problem Scenarios

When `broadcastCompactBlock` is called from the `handleCompactBlock` function (code ref↗), a failure in `TrySendEnvelopeShim` prevents the compact block from being propagated to peers.

Recommendation

We recommend replacing silent failures with proper error handling and implementing the retry mechanism.

Notably, the clients were aware of the implications of using the `TrySendEnvelopeShim` function prior to the start of the audit.

TrySendEnvelopeShim can silently fail in broadcastHaves function

Severity Low **Impact** 1 - Low **Exploitability** 1 - Low
Type Implementation **Status** Acknowledged

Involved artifacts

- `consensus/propagation/commitment.go`↗
- `consensus/propagation/have_wants.go`↗

Description

The `broadcastHaves` function doesn't handle `TrySendEnvelopeShim` failure (code ref↗). Additionally, haves will incorrectly be added to the peer's `haves` (code ref↗).

Problem Scenarios

When `broadcastHaves` is called from the `sendWantsThenBroadcastHaves` function (code ref↗), a failure in `TrySendEnvelopeShim` prevents the part from being propagated to peers. Correspondingly, in the `recoverPartsFromMempool` function, a silent failure of `TrySendEnvelopeShim` could go unregistered (code ref↗).

Furthermore, if `TrySendEnvelopeShim` fails within `broadcastHaves` when invoked by `handleRecoveryPart`, other peers will not receive the `HaveParts` message from the node that already possesses all the parts (code ref↗).

Recommendation

We recommend replacing silent failures with proper error handling and implementing the retry mechanism, as mentioned in the comment (ref↗).

Notably, the clients were aware of the implications of using the `TrySendEnvelopeShim` function prior to the start of the audit.

GetPart returning nil causes a runtime panic

Severity Informational **Impact** 3 - High **Exploitability** 0 - None

Type Implementation **Status** Acknowledged

Involved artifacts

- `consensus/propagation/have_wants.go`↗
- `consensus/propagation/commitment.go`↗

Description

In several places throughout the code, when calling the `GetPart` method, it is not checked whether the returned part is nil (e.g., code ref↗, code ref↗, code ref↗). In the case it is nil, it will trigger a runtime panic.

Problem scenarios

This is unlikely to happen unless the internal state is corrupted.

Recommendation

Our recommendation is to explicitly handle the case when `GetPart` returns nil.

Discrepancy between the implementation and specification

Severity Informational **Impact** 1 - Low **Exploitability** 0 - None
Type Documentation **Status** Risk Accepted

Involved artifacts

- Specifications for high throughput recovery↗
- ADR for high throughput recovery↗
- Implementation

Description

We identified that there are some differences between the protocol specification and the implementation. The Celestia team is already aware of these deviations and the related attack vectors, and remediation work is in progress. However, until the implementation is fully aligned with the specification, these discrepancies continue to constitute technical debt and understanding the core of the implemented protocol requires reverse-engineering of the code, especially for the catch-up mechanism.

Peer disconnection is not implemented

The protocol description mentions some disconnection rules to disconnect from peers that exhibit malicious behaviors. However, these rules are not currently implemented and this might have security repercussions as explained in the finding ***The specified disconnection rules were not implemented***.

Compact block signatures are not implemented

According to the spec, each compact block must carry a signature to guard against tampering. The current implementation uses random bytes as a placeholder for the signatures. In addition, the handling of the signatures in the case of a compact block introduced via the catch-up mechanism should be addressed.

Catch-up mechanism lacking specification

The catch-up mechanism is an integral part of the system, interacting with the main PBBT propagation protocol and the consensus reactor. It should be precisely specified especially where and why it is invoked by the consensus reactor and its impact on the internal state of the propagation reactor.

Broadcast Tree Security Assumptions

According to the specification, securing the system requires both parity data and a validator overlay network. While a full mesh overlay is not yet implemented, the clients indicated they plan to move forward without blocking on its completion.

Incorrect Assumptions in Request Limits

As noted by the clients prior to audit, the `PerPeerConcurrentRequestLimit` is currently calculated under the assumption that voting power is equally distributed among validators.

Recommendation

A precise and complete documentation of all parts of the protocol, their expected behaviors and interactions helps onboarding on the code, facilitates future changes in the code and will expedite future audits.

TxMetaData cannot be validated before block is complete

Severity Informational **Impact** 1 - Low **Exploitability** 0 - None
Type Protocol **Status** Risk Accepted

Involved artifacts

- `specs/src/recovery.md`↗
- Definition DD3 of the threat model

Description

By design of the data structures and protocol, the `TxMetaData` of a `CompactBlock` information - different to the other data structures in the protocol - cannot be validated (e.g. with the compact block) until the block is reconstructed and CometBFT-validated. This is not about signature (which proves the origin of the message) but a malformed message from a malicious proposer.

Problem Scenarios

An earlier detection would avoid running the full protocol for an invalid proposal message.

Recommendation

We recommend validating the `TxMetaData` when redesigning the data structures during the implementation of the planned changes.

Nodes don't drop duplicate WantParts message

Severity Informational **Impact** 0 - None **Exploitability** 3 - High

Type Protocol **Status** Acknowledged

Involved artifacts

- `consensus/propagation/have_wants.go`↗
- `specs/src/recovery.md`↗

Description

According to the spec↗, "Nodes MUST avoid sending the same `Have` or `Want` message to the same peer more than once". This is consistent with property AL14 in the threat model. However, There is no explicit protection against receiving duplicate `WantParts` messages in the implementation (code ref↗). Thus, the spec and the implementation are not consistent. In practice, this is fine due to per peer bandwidth constraints and because if the part is valid, the node can safely handle it.

Problem scenarios

This issue can occur any time a node sends to its peers valid duplicate `WantParts` messages.

Recommendation

Our recommendation is to update the spec to be consistent with the implementation.

Nodes don't drop RecoveryPart message if they were not requested

Severity Informational **Impact** 0 - None **Exploitability** 3 - High

Type Protocol **Status** Acknowledged

Involved artifacts

- `consensus/propagation/have_wants.go`↗
- `specs/src/recovery.md`↗

Description

According to the spec↗, "*Nodes MUST only send `Data` messages if that data has been requested*". This is consistent with property AL8 in the threat model. However, in the implementation, when receiving a `RecoveryPart` message, a node doesn't check if it previously sent a request for that part (code ref↗). Thus, the spec and the implementation are not consistent. In practice, this is probably fine as if the part is valid, the node can safely handle it since it already received it.

Problem scenarios

This issue can occur any time a node sends to its peers valid `RecoveryPart` messages without first receiving `WantParts` messages.

Recommendation

Our recommendation is to update the spec to be consistent with the implementation.

Miscellaneous code findings

Severity Informational **Impact** 0 - None **Exploitability** 0 - None
Type Implementation **Status** Acknowledged

Description

In this finding, we describe a number of improvements to the code. Those typically do not affect the functionality, but improve the code readability, make code more robust with respect to future changes, or represent a good engineering practice.

1. Typo in the comment (code ref↗).
2. This check↗ is already done in `ValidateBasic`↗ when validating the proof.
3. Minor typo in the comment (code ref↗).
4. All the changes in `NewPartSetFromData` are unnecessary (code ref↗).
5. in `TestEncoding`, use data that is not a multiple of `testPartSize` to test the padding (code ref↗).
6. The catchup field in `proposalData` is never used (code ref↗).
7. Throughout the code, the value 2 is used as the parity ratio instead of the defined constant `ParityRatio` (e.g., `NewCombinedSetFromCompactBlock`↗, `AddCommitment`↗, `MissingOriginal`↗).
8. It is not safe to call the `DecreaseConcurrentReqs` method concurrently for the same peer as it might result in a negative `concurrentReqs` (code ref↗).
9. The `haves` bit array in the peer state is never used, only updated (code ref↗).
10. Many of the bit array methods could return nil, but this is rarely checked in the code (code ref↗) vs (code ref↗). This might lead to a panic.
11. Calling the `Encode` method with `ops = nil` will cause a runtime panic (code ref↗).
12. Minor typo in the comment (code ref↗).
13. Minor typo (code ref↗).
14. `Signature` is used as a proxy for catchup in `SyncData` (code ref↗).
15. The propagation reactor doesn't check if a peer supports propagation before adding them (code ref↗), similar to here↗.
16. Replacing the `&&` with `||` here↗.
17. The `GetTrueIndices` function (code ref↗) can panic due to nil pointer dereference when `bA == nil` or array bounds errors when `len(bA.Elems) == 0`, as it lacks defensive checks before accessing `bA.Elems` and `bA.Bits`. While the team has not identified any current usage of this function that would lead to such issues, adding nil and empty slice validation at the function start would prevent these runtime panics in future development.
18. `isProposalComplete` function in `syncData` is incorrectly used. In the consensus reactor, the `syncData` routine calls `isProposalComplete` (code ref↗) to check if the current proposal block was properly received. However, in `isProposalComplete`, for a proposal to be considered complete, the node must have both the proposal and its associated block, and if a `POLRound` was set, it must have received a two-thirds majority of pre-votes from that round (code ref↗). The second required condition can cause a proposal to fail the completeness check (code ref↗) even though all the block parts are available and the block is complete.

Appendix: Vulnerability classification

For classifying vulnerabilities identified in the findings of this report, we employ the simplified version of Common Vulnerability Scoring System (CVSS) v3.1↗, which is an industry standard vulnerability metric. For each identified vulnerability we assess the scores from the *Base Metric Group*, the Impact score↗, and the Exploitability score↗. The *Exploitability score* reflects the ease and technical means by which the vulnerability can be exploited. That is, it represents characteristics of the *thing that is vulnerable*, which we refer to formally as the *vulnerable component*. The *Impact score* reflects the direct consequence of a successful exploit, and represents the consequence to the *thing that suffers the impact*, which we refer to formally as the *impacted component*. In order to ease score understanding, we employ CVSS Qualitative Severity Rating Scale↗, and abstract numerical scores into the textual representation; we construct the final *Severity score* based on the combination of the Impact and Exploitability sub-scores.

As blockchains are a fast evolving field, we evaluate the scores not only for the present state of the system, but also for the state that deems achievable within 1 year of projected system evolution. E.g., if at present the system interacts with 1-2 other blockchains, but plans to expand interaction to 10-20 within the next year, we evaluate the impact, exploitability, and severity scores wrt. the latter state, in order to give the system designers better understanding of the vulnerabilities that need to be addressed in the near future.

Impact Score

The Impact score captures the effects of a successfully exploited vulnerability on the component that suffers the worst outcome that is most directly and predictably associated with the attack.

ImpactScore	Examples
High	Halting of the chain; loss, locking, or unauthorized withdrawal of funds of many users; arbitrary transaction execution; forging of user messages / circumvention of authorization logic
Medium	Temporary denial of service / substantial unexpected delays in processing user requests (e.g. many hours/days); loss, locking, or unauthorized withdrawal of funds of a single user / few users; failures during transaction execution (e.g. out of gas errors); substantial increase in node computational requirements (e.g. 10x)
Low	Transient unexpected delays in processing user requests (e.g. minutes/a few hours); Medium increase in node computational requirements (e.g. 2x); any kind of problem that affects end users, but can be repaired by manual intervention (e.g. a special transaction)

ImpactScore	Examples
None	Small increase in node computational requirements (e.g. 20%); code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation

Exploitability Score

The Exploitability score reflects the ease and technical means by which the vulnerability can be exploited; it represents the characteristics of the vulnerable component. In the below table we list, for each category, examples of actions by actors that are enough to trigger the exploit. In the examples below:

- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be
  - *legitimate*, e.g. submission of a transaction that follows protocol rules by a user; delegation/redelegation/bonding/unbonding; validator downtime; validator voting on a single, but alternative block; delays in relaying certain messages, or speeding up relaying other messages;
  - *illegitimate*, e.g. submission of a specially crafted transaction (not following the protocol, or e.g. with large/incorrect values); voting on two different alternative blocks; alteration of relayed messages.
- We employ also a *qualitative measure* representing the amount of certain class of power (e.g. possessed tokens, validator power, relayed messages): *small* for < 3%; *medium* for 3-10%; *large* for 10-33%, *all* for >33%. We further quantify this qualitative measure as relative to the largest of the system components. (e.g. when two blockchains are interacting, one with a large capitalization, and another with a small capitalization, we employ *small* wrt. the number of tokens held, if it is small wrt. the large blockchain, even if it is large wrt. the small blockchain)

ExploitabilityScore	Examples
High	illegitimate actions taken by a small group of actors; possibly coordinated with legitimate actions taken by a medium group of actors
Medium	illegitimate actions taken by a medium group of actors; possibly coordinated with legitimate actions taken by a large group of actors
Low	illegitimate actions taken by a large group of actors; possibly coordinated with legitimate actions taken by all actors
None	illegitimate actions taken in a coordinated fashion by all actors

Severity Score

The severity score combines the above two sub-scores into a single value, and roughly represents the probability of the system suffering a severe impact with time; thus it also represents the measure of the urgency or order in which vulnerabilities need to be addressed. We assess the severity according to the combination scheme represented graphically below.

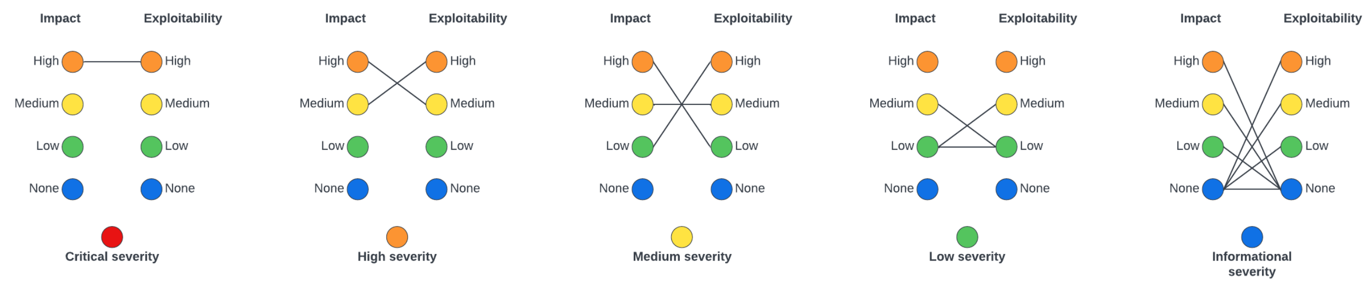

Figure 1: Severity classification

As can be seen from the image above, only a combination of high impact with high exploitability results in a Critical severity score; such vulnerabilities need to be addressed ASAP. Accordingly, High severity score receive vulnerabilities with the combination of high impact and medium exploitability, or medium impact, but high exploitability.

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability, etc.) set forth in the associated Services Agreement. This report provided in connection with the Services set forth in the Services Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement.

This audit report is provided on an “as is” basis, with no guarantee of the completeness, accuracy, timeliness or of the results obtained by use of the information provided. Informal has relied upon information and data provided by the client, and is not responsible for any errors or omissions in such information and data or results obtained from the use of that information or conclusions in this report. Informal makes no warranty of any kind, express or implied, regarding the accuracy, adequacy, validity, reliability, availability or completeness of this report. This report should not be considered or utilized as a complete assessment of the overall utility, security or bugfree status of the code.

This audit report contains confidential information and is only intended for use by the client. Reuse or republication of the audit report other than as authorized by the client is prohibited.

This report is not, nor should it be considered, an “endorsement”, “approval” or “disapproval” of any particular project or team. This report is not, nor should it be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts with Informal to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor does it provide any indication of the client’s business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should it be leveraged as investment advice of any sort.

Blockchain technology and cryptographic assets in general and by definition present a high level of ongoing risk. Client is responsible for its own due diligence and continuing security in this regard.