# Security Audit Report

Skip: mev-tendermint

Authors: Manuel Bravo, Mirel Dalcekovic

Last revised 11 April, 2023

## Table of Contents

# Audit Overview

## Scope of this report

This is a report on the code review audit of Skip's mev-tendermint https://github.com/skip-mev/mev-tendermint/releases/tag/v0.34.24-mev.14

The audit took place from February 3, 2023 through March 14, 2023 by Informal Systems by the following personnel:

- Manuel Bravo
- Mirel Dalcekovic

The scope of this audit was limited to manual code review and manual analysis and reconstruction of the protocols.

## Conducted work

The audit team started its work on February 3rd. We have had weekly meetings with the Skip team to ask questions, confirm our understanding and present findings. The development team gave us all the necessary documentation (videos and documents) to ease the understanding of the Skip protocol.

The audit team engaged in reviewing the changes made by Skip to two main components of Tendermint:

- Changes made to the p2p package to allow Skip validators to register with their bundle relayer (the sentinel), receive bundles of transactions that are treated atomically, and reconnect if they lose connection.
- Changes made to the mempool and the reaping functions to allow validators to reap Skip bundles at the top of their block.

We have also written an English specification of the protocol. This work included formalizing assumptions and formulating desired properties, as well as analyzing which of these properties are guaranteed by the protocol.

The sentinel's code was not audited.

## Conclusions

Overall, we found the codebase to be of good quality: the code is well-structured and easy to follow. Our main concern was to check whether there was a chance that consensus is delayed in any way due to changes introduced to both the p2p and the mempool layers. We concluded that it is not the case at the cost of weakening traders' guarantees. In the current design, these two desired properties are in tension.

Despite the general good quality, we found some details that should be addressed in order to raise the quality of the code, although none critical.

## Disclaimer

# System Overview

## Synopsis

This section specifies the data structures and subprotocols of the Skip protocol. The Skip protocol implements an auction system that allows Cosmos users to capture maximal extractable value (MEV) across a variety of chains in the ecosystem.

## Content

- Motivation and Basic Concepts
- System Model and Properties
- Technical Specification
- Properties Analysis

## Limitations of the Specification

- The implementation maintains a legacy sidecar channel. This is not detailed in this specification.
- Metrics and logging systems are omitted from the specification.
- The logic around `notifyTxsAvailable` is omitted from the specification.
- The API of `SyncMap` (this spec) and `sync.map` (the implementation) is slightly different. For instance, `sync.map` offers a `LoadOrStore` method that `SyncMap` does not. If relevant, this could be added in the future.

## Further Reading

- Skip's white paper
- About Tendermint's `timeout_commit`
- About cross-block transaction duplication

# Motivation and Basic Concepts

## Motivation

Maximal extractable value (MEV) refers to a strategy to include, omit, or reorder transactions when creating a block to attempt to maximize profit. For instance, a block proposer can trivially censor, reshuffle, or inject their own transactions to maximize its own profit as far as the block includes valid transactions. A second, more disturbing, example of MEV attacks in the Cosmos ecosystem are orchestrated by users (aka traders). Currently, Cosmos chains like Juno, Terra and Osmosis have a first-come-first-serve mempool. Users exploit this fact to win arbitrage opportunities by locating their arbitrage bots as close to as many validators and full nodes as possible and spam them with transactions. This has many negative effects such as centralization of profits and chain halting risk. See this for more information about the negative effects.

Due to the limitations of the current ABCI interface, preventing block proposers to capture MEV is very challenging. The Skip protocol instead is concerned with protecting users from the harmful effects of MEV in the second scenario. Skip's goal is to allow traders to compete in a fair auction system and eliminate forms of MEV that are toxic, such as sandwiching and spam.

## Definitions

A `mev-validator` is a validator running the skip protocol.

A `bundle` is an ordered set of transactions.

A `mev-transaction` is a transaction received from a sentinel node. Each `mev-transaction` must belong to a bundle.

The `sidecar` is a mempool for bundle instantiated by mev-validators.

A bundle is complete at a mev-validator when its sidecar mempool stores all its containing transactions.

We use `mempool` to refer to Tendermint's mempool.

A `trader` (aka searcher) is a participant of an auction. Traders participate in auctions by submitting bundles to sentinel nodes.

`Sentinel nodes` are p2p nodes that run the auction. Sentinel nodes receive bundles from traders, append a payment transaction to the bundle and forward them to validators in an order that respects the associated fees. There is a single sentinel node per chain. We use chain sentinel to refer to the sentinel of a given chain.

## System Model and Properties

### Assumptions

#### Assumption 1: Sentinel nodes are trustworthy

Validators assume that sentinel nodes follow the protocol. This implies that:

- Sentinel nodes do not censor or reshuffle transactions in the bundles submitted by traders.

#### Assumption 2: Weak network guarantees

Validators do not expect any guarantee from the network. This means that messages sent from the sentinel may take indefinitely long, get lost or be reordered.

### Desired Properties

#### Property 1 [No-disruption]: Minimal consensus disturbance

This rules out designs that require synchronous communication with an external component during consensus operation (aka pull-based models) in favor of designs that distribute bundles in the background, out of consensus operation critical path (aka push-based models).

> *For instance, if the proposer of a block engages into a synchronous communication with its sentinel node to retrieve a set bundles during block creation, the property would be violated. This is because block creation would be delayed which may affect the correct operation of consensus: timer management would need to account for the block creation delay.*

#### Property 2 [Atomicity]: Atomicity for bundles of transactions

Bundles are an indivisible and irreducible set of transactions such that when a bundle is included in a block either all transactions are executed or none.

#### Property 3 [Order]: The transactions within a bundle are executed in order

The set of transactions within a bundle are totally ordered and must be executed in that order.

#### Property 4 [At-most-once]: A block should not include duplicates

Transactions are not always idempotent, thus including duplicated transactions in a block should be prevented.

> *Bundles typically include transactions from the mempool. Thus, when blocks include transactions from both the mempool and the sidecar, it is likely to have duplicates. To satisfy this property, one has to remove duplicates.*

#### Property 5 [Priority]: Priority guaranteed for highest paying bundles

Assume that for a given action the sentinel has received two bundles `b1` and `b2` such that the trader that submitted `b1` is willing to pay a higher fee. By this property, the protocol should guarantee that if only one bundle is included in the auctioned block, then it has to be `b1` : `b1` has priority over `b2` . The property also allows both bundles to be included or none.

### Property 6 [Optimal-reaping]: A proposer includes in as block as many bundles as possible

The protocol must guarantee that a block proposer reaps as many bundles from the sidecar mempool as possible, given the constraints of other properties such as Priority and Atomicity. This is important to maximize profit.

### Property 7 [Privacy]: Privacy for users submitting bundles

It is important that transactions submitted by traders and the fees they are willing to pay are not visible to other traders or validators until the transactions are committed, e.g., this is important to minimize front-running.

### Property 8 [Accountability]: Cheater traders are ejected

If a trader submits a front-running or a sandwich bundle it must be ejected.

# Technical Specification

## General Design

The Skip protocol integrates two main components: the sidecar mempool and the sentinel. Each mev-validator instantiates the sidecar mempool locally and connects to the sentinel of the chain it validates through the p2p layer. The protocol takes the following steps:

1. Traders submit bundles to the sentinel. Submitted bundles are typically composed of two transactions: a transaction that the trader has seen in its local Tendermint mempool and a second transaction that must be executed after the first. Traders have access to the Tendermint mempool because they usually run a full node. Together with each bundle, the trader submits two key pieces of data:
    - A height `desiredHeight` that indicates the height at which the bundle must be included in the chain. When the trader sets `desiredHeight` to `0`, this indicates the sentinel that the bundle should be included in the next height.
    - A fee that the trader is willing to pay if the bundle makes it to the block at the desired height.
2. The sentinel accepts bundles for a given height until `400ms` before the first proposer of that height starts creating the block. At that point, we say that the sentinel closes the auction for that height. The sentinel infers this time by starting a timer when it learns that the previous height has been decided based on Tendermint's `timeout_commit`: how long we wait after committing a block, before starting on the new height. Of course, this is not perfectly synchronized: the time that the first proposer of the height starts its `timeout_commit` is not necessarily the time when the sentinel learns about the previous decision.
3. Once the auction for a height is closed, the sentinel starts sending bundles to the first proposer of that height in descending fee ordered: first those bundles with higher fees. The sentinel sends bundles in parts, each including a single transaction called `SidecarTx`. This is a limitation of the p2p layer of Tendermint: it is designed to include a single transaction per p2p message. It is worth noting that the sentinel appends a transaction to each of the bundles with a payment transaction.
4. When a mev-validators receives a `SidecarTx` calls the `AddTx` function, which after some safety checks, adds the transaction to its sidecar mempool.
5. When a mew-validator becomes the block proposer, it reaps both the sidecar mempool and Tendermint's mempool to create the block. It first reaps the sidecar mempool looking for fully received bundles, respecting the order fixed by the sentinel. Then if there is room for more transactions in the block, it includes transactions from Tendermint's mempool.
6. After a block is committed, the mev-validator updates the sidecar mempool by calling the `Update` function. This function removes from the sidecar mempool any `sidecarTx` belonging to a bundle whose desired height has already passed.

## Data Structures

The protocol does not assume any specific transaction format. It assumes that a transaction `Tx` is a byte array `byte[]` of any size. Transactions are uniquely identified by a key `TxKey`, a byte array `byte[]` of fixed size. Typically, the key of a transaction is its hash. We assume that a function `Key(tx: Tx)` returns the transaction's key.

Throughout the specification, we use a few data types that are not standard and have not been introduced:

- `SyncMap` is a thread-safe `Map`.
- `SyncArray` is a thread-safe `Array`. We assume that there is a method `delete(elem)` that removes `elem` from the array and shifts its elements accordingly.

- `Cache` implements a cache for transactions. The cache defines a maximum size. Once the maximum is reached, elements are dropped when new elements are added. The cache offers the following API:
  - `Reset()` : resets the cache to an empty state.
  - `Push(tx: Tx)` : checks if the transaction is in the cache. If it is already, it returns `false`. Otherwise, it adds the transaction and returns `true`.
  - `Remove(tx: Tx)` : removes the transaction from the cache.
  - `Has(tx: Tx)` : reports whether the transaction is present in the cache or not.
- `Mutex` and `RWMutex` are self-explanatory.
- `Error` is left undefined intentionally. The error variable name should be self-explanatory.

A `SidecarTx` is a transaction that belongs to a bundle. Mev-validators store these in their local sidecar mempool.

```
interface SidecarTx {
    tx: Tx // tx bytes
    desiredHeight: int
    bundleId: int
    bundleOrder: int
    bundleSize: int
    gasWanted: int
    senders: SyncMap<int, bool>
}
```

- `tx` is the transaction.
- `bundleOrder` defines the order of the transaction within the associated bundle.
- `desiredHeight` and `bundleId` are as defined above.
- `bundleSize` is the same than `enforcedSize` for bundles.
- `gasWanted` determines the amount of gas executing the transaction would require.
- `senders` is a thread-safe map that tracks from which peers the transactions has been received. It goal is to avoid broadcasting transactions to peers that already have them.

Bundle is defined as follows.

```
interface Bundle {
    orderedTxsMap: SyncMap<int, SidecarTx>
    desiredHeight: int
    bundleId: int
    currentSize: int
    enforcedSize: int
}
```

- `orderedTxsMap` stores the set of transactions that belong to the bundle that have been received.
- The `desiredHeight` determines the height at which the bundle must be included in a block.
- For a given height, `bundleId` defines the order of this bundle relative to all the bundles received by the sentinel during the auction for `desiredHeight`.
- `currentSize` determines the total number of transactions stored in `orderedTxsMap`.
- `enforcedSize` defines the total size of the bundle.

Bundles are uniquely identified by an integer id `bundleId` and its desired height `desiredHeight`. It is the sentinel who is in charge of enforcing the uniqueness of bundle ids.

```
interface BundleKey {
    desiredHeight: int
    bundleId: int
}
```

The Sidecar is defined as follows:

```
interface Sidecar {
    height: int
    heightForFiringAuction: int
    txsBytes: int
    lastBundleHeight: int
    txs: SyncArray
    txsMap: SyncMap<TxKey, Tx>
    bundles: SyncMap<BundleKey, Bundle>
    maxBundleID: int
    updateMtx: RWMutex
    maxBundleIDMtx: Mutex
    bundleSizeMtx: Mutex
    cache: Cache
}
```

- `height` is the latest committed height.
- `heightForFiringAuction` is the next height to be committed.
- `txsBytes` is the total size of sidecar in bytes.
- `lastBundleHeight` is height of last accepted bundle.
- `txs` is a threat-safe ordered list. It is primarily used by the p2p layer to broadcast SidecarTx to other peers.
- `txsMap` stores `SidecarTx` transactions in a thread-safe map for fast access.
- `bundles` stores bundles in a thread-safe map. Each bundle then stores the set of `SidecarTx` that compose it in its `orderedTxsMap` map.
- `updateMtx`, `maxBundleIDMtx` and `bundleSizeMtx` are mutexes to manage the concurrent access to the sidecar mempool.
- `cache` keeps a cache of already-seen txs.

Finally, `ReapedTxs` is a pair including a list of transactions and one mapping the gas estimation required to execute each of the transactions. This is used by `CombineSidecarAndMempoolTxs` and `ReapMaxTxs`.

```
interface ReapedTxs {
    txs: Tx[]
    gasWanteds: int[]
}
```

## The P2P Layer Protocols

This section describes how the main components of the Skip protocol fit within the p2p layer. We assume the most common case in which mev-validators are not reachable directly from sentinels: this is a common deployment where validators are shielded from attacks, e.g., DoS attacks, by an infrastructure of sentry nodes. In the case that there is no sentry nodes infrastructure, the mev-validator will be connected to the chain sentinel directly. Thus, the following description of how sentry nodes connect to the chain sentinel would apply to mev-validators.

### Configuration

Mev-validators, sentry nodes and sentinels are all nodes of the p2p layer. As mentioned above, there is one sentinel per chain. The sentinel node communicates directly with the sentry nodes via a special channel `sidecarChannel`. This channel is secured using the same authentication handshake as the one used by Tendermint to secure all the other forms of p2p communication. For a sentry node, its chain sentinel node realizes multiple Tendermint peer roles:

1. It is a persistent peer. This means that it should maintain a connection with it at all times.
2. It is a private peer. This means that its identity should not be disclosed to other p2p nodes.
3. It is an unconditional peer. This means that the connection to the sentinel node will always be maintained and reestablished in case of disconnection.

Note that not all sentry nodes necessarily connect to the chain sentinel. For instance, one can build a topology in which only one sentry node connects to the chain sentinel and only one connects to the mev-validator. This way operators can decide how `SidecarTx` flow within its infrastructure. To do so, sentry nodes can define a set of personal peers. When a sentry node receives a `SidecarTx`, it broadcasts it to its set of personal peers (omitting those from which it has received it). The operator should guarantee that the mev-validator receives `SidecarTxs` in time: before the mev-validator creates its block when it is the first proposer of a consensus instance.

A sentry node that adds the chain sentinel to its set of persistent peers connects to it upon starting. This is automatically managed by the p2p layer: p2p nodes try to connect to the list of persistent peers automatically upon starting. Sentry nodes should always try to reconnect to the chain sentinel in case it gets disconnected (or simply did not manage to establish a connection on startup).

> *Note that the implementation treats sentinel nodes slightly different than the other persistent peers in which there is there is no exponential back-off, i.e., retry interval does not grow exponentially over time. One must be careful to avoid retrying to reconnect too often such that reconnection logic is too expensive.*

### A `SidecarTx` Throughout the P2p Layer

A `SidecarTx` may take the following steps from the moment a chain sentinel sends it until it reaches a mev-validator:

- The chain sentinel sends a message with the `SidecarTx` to the set of sentry nodes to which it has a direct connection. The expected format of the sentinel messages is as defined in the interface `SidecarTx` introduced above.
- When a sentry node receives such a message, it first must check that the sender is the sentinel or a personal peer. If it is not coming from one of these nodes, the message is ignored. Otherwise, it is accepted.
- When the message is accepted, the node tries to add it to its sidecar mempool by calling the `AddTx` function (described below).
- Those transactions added to the sidecar mempool are broadcast to other personal peers through the `sidecarChannel`. As mentioned above, the goal is to eventually reach the mev-validator.

## The Sentinel Protocols

> *NOTE: Out of the scope of this audit. TO BE COMPLETED.*

## The Sidecar Protocols

In this section, we describe the main functionality of the sidecar: `AddTx`, `Update`, `ReapMaxTxs` and `CombineSidecarAndMempoolTxs`. The remaining are either never referenced or just used for testing purposes and are therefore omitted from the specification.

When a mev-validator receives a mev-transaction through a valid channel, it calls the `AddTx` function. The function takes the following steps:

- It first acquires a read lock. This is because `AddTx` could be called concurrently.
- Discards the transaction in any of the following cases:
  - If the transaction is already in the cache. The cache is cleaned after committing a block in the `Update` function. Thus this guarantees that the transaction is not added if it was already received after the last time the cache was cleaned. This can happen due to the weak network assumptions (Assumption 2), despite the sentinel behaving correctly (Assumption 1).
  - If the transaction belongs to a bundle whose desired height is less than the next firing auction height. This can happen due to the weak network assumptions (Assumption 2), despite the sentinel behaving correctly (Assumption 1).
  - If the transaction is assigned an order within the bundle that is greater than the bundle's size. By Assumption 1, this should never happen.
- Retrieves the bundle from `sidecar.bundles` (or creates it if it does not exist). The function discards the transaction in any of the following cases:
  - If there is a mismatch between the bundle size stored at the sidecar mempool and the one carried by the transaction. By Assumption 1, this should never happen.
  - If the bundle is already full.
  - If the sidecar has already added a transaction to the bundle in that position.
- Creates the `SidecarTx` and adds it to the bundle's `orderedTxsMap`.
- Updates `maxBundleID` if it is the greatest seen so far.
- Finally, it adds the `SidecarTx` to the sidecar's `txs` and `txsMap`, updates `sidecar.txsBytes` and `sidecar.lastBundleHeight`.

```
function AddTx(
    sidecar: Sidecar,
    tx: Tx,
    senderId: int,
    desiredHeight: int,
    bundleSize: int,
    bundleOrder: int,
    bundleId: int,
    gasWanted: int
    ): Error {
    sidecar.updateMtx.RLock()

    // check if the transaction is in the cache
    if (sidecar.cache.Push(tx) === false) {
        // register the sender if the transaction is still in the map
        txKey = Key(tx)
        if (sidecar.txsMap.has(txKey)) {
            scTx = sidecar.txsMap.get(txKey)
            scTX.senders.set(senderId, true)
        }
        // return already-in-cache error
        sidecar.updateMtx.RUnlock()
        return ErrTxInCache
    }

    // discard tx if the desiredHeight has already passed
    if (desiredHeight < sidecar.heightForFiringAuction) {
        sidecar.updateMtx.RUnlock()
        return ErrWrongHeight
    }

    // discard tx if asking to be included has an order greater/equal to the bundle
size
    // prevented by Assumption 1
    if (bundleOrder >= bundleSize) {
        sidecar.updateMtx.RUnlock()
        return ErrTxMalformedForBundle
    }

    // retrieve or create bundle
    bundleKey = BundleKey{desiredHeight, bundleId}
    if (sidecar.bundles.has(bundleKey)) {
        bundle = sidecar.bundlers.get(bundleKey)
    } else {
        bundle = Bundle{desiredHeight: desiredHeight,
                        bundleID: BundleId,
                        currentSize: 0,
                        enforcedSize: bundleSize,
                        orderedTxsMap: new Map<> }
    }

    // prevented by Assumption 1
    if (BundleSize !== bundle.EnforcedSize) {
        sidecar.updateMtx.RUnlock()
        return ErrTxMalformedForBundle
    }

    sc.bundleSizeMtx.Lock()

    if (bundle.currentSize === bundle.enforcedSize) {
        sidecar.bundleSizeMtx.Unlock()
        sidecar.updateMtx.RUnlock()
        return ErrBundleFull
    }

    if (bundle.orderedTxsMap.has(bundleOrder)) {
        sidecar.bundleSizeMtx.Unlock()
        sidecar.updateMtx.RUnlock()
        return nil
    }

    // create sidecar tx
    scTx = SidecarTx{desiredHeight: desiredHeight,
                       tx:            tx,
                       bundleID:      bundleID,
                       bundleOrder:   bundleOrder,
                       bundleSize:    bundleSize,
                       gasWanted:     gasWanted }

    bundle.orderedTxsMap.set(bundleOrder, scTx)

    bundle.currentSize = bundle.currentSize + 1

    sidecar.bundleSizeMtx.Unlock()

    sidecar.maxBundleIDMtx.Lock()
    if (bundleId > sidecar.maxBundleId) {
        sidecar.maxBundleId = bundleId
    }
    sidecar.maxBundleIDMtx.Unlock()

    sidecar.txs.push(scTx)
    txKey = Key(scTx.tx)
    sidecar.txsMap.set(txKey, scTx)

    // this is done atomically in the implementation
    sidecar.txsBytes = sidecar.txsBytes + len(scTx.tx)

    sidecar.lastBundleHeight = scTx.desiredHeight

    sidecar.updateMtx.RUnlock()

    return nil
}
```

The `Update` function is called by the consensus when a block is committed.

- It first updates `sidecar.height` and `sidecar.heightForFiringAuction`.
- It then iterates over the committed transactions and removes them from `sidecar.txs` and `sidecar.txs` by calling `RemoveTx`.
- It sets `sidecar.maxBundleID` to zero and resets the cache.
- Then it iterates over all transactions in `sidecar.txs` and removes those whose desired height is less or equal to the just committed height.
- Finally, it removes all bundles from `sidecar.bundles` whose desired height has passed, and cannot ever be committed.

```
function Update(
    sidecar: Sidecar,
    height: int,
    txs: Tx[]
    ) {

    sidecar.height = height
    sidecar.notifiedTxsAvailable = false
    sidecar.heightForFiringAuction = height + 1
    for (let tx of txs) {
        txKey = Key(tx)
        if sidecar.txsMap.has(txKey) {
            RemoveTx(sidecar, tx)
        }
    }

    sidecar.cache.Reset()
    sidecar.maxBundleID = 0

    for (let scTx of sidecar.txs) {
        if scTx.desiredHeight <= height {
            RemoveTx(sidecar, tx)
        }
    }

    for (let key of sidecar.bundles.keys())
        bundle = sidecar.bundles.delete(key)
        if bundle.desiredHeight <= height {
            sidecar.bundles.delete(key)
        }
}
```

The `RemoveTx` function is only called by the `Update` function to remove a transaction from both `sidecar.txs` and `sidecar.txsMap`, as well as to update `sidecar.txsBytes`.

```
function RemoveTx(
    sidecar: Sidecar,
    tx: Tx
) {
    sidecar.txs.delete(tx)
    txKey = Key(tx)
    sidecar.txsMap.delete(txKey)
    sidecar.txsBytes = sidecar.txsBytes - len(tx)
}
```

The `ReapMaxTxs` function is called by consensus when a proposer prepares its proposal.

- It iterates over all bundles with `bundleKey = BundleKey{sidecar.heightForFiringAuction, bundleId}` from `bundleId=0` up to `sidecar.maxBundleId`.
- It reaps in `bundleId` order each of the existing bundles that are complete by adding the containing transactions into an ordered list `scTxs`.
- Finally, it returns two lists, one with the list of transactions and one mapping the gas estimation required to execute each of the transactions.

```
function ReapMaxTxs(sidecar: Sidecar): ReapedTxs {
    sidecar.updateMtx.RLock()

    if (sidecar.txs.length == 0) || (sidecar.bundles.size == 0) {
        sidecar.updateMtx.RUnlock()
        return ReapedTxs{[], []}
    }
    scTxs = SidecarTx[]

    for (let bundleId = 0; bundleId <= sidecar.maxBundleId; bundleId++) {
        bundleKey = BundleKey{sidecar.heightForFiringAuction, bundleId}
        if sidecar.bundles.has(bundleKey) {
            bundle = sidecar.bundles.get(bundleKey)
            // ignore bundle if not complete
            if (bundle.currentSize !== bundle.enforcedSize) {
                continue
            }
            innerTxs = SidecarTx[]
            // at this point the bundle exists and it is complete
            for (bundleOrderIter = 0; bundleOrderIter < bundle.EnforcedSize;
bundleOrderIter++) {
                if (bundle.orderedTxsMap.has(bundleOrderIter)) {
                    scTx = bundle.orderedTxsMap.get(bundleOrderIter)
                    innerTxs = innerTxs.push(scTx)
                } else {
                    break
                }
            }

            if (innerTxs.length !== bundle.EnforcedSize) {
                scTxs = scTxs.push(innerTxs)
            }
        }
    }

    txs = Tx[]
    gasWanted = int[]

    for (let scTx of scTxs) {
        txs = txs.push(scTx.tx)
        gasWanted = gasWanted.push(scTx.gasWanted)
    }

    sidecar.updateMtx.RUnlock()

    return ReapedTxs{txs, gasWanted}
}
```

## Block Creation Logic

The proposer of a block first reaps bundles from the sidecar mempool by calling `ReapMaxTxs` and then from Tendermint's mempool by calling `ReapMaxBytesMaxGas`. It then merges both sets of transactions by calling `CombineSidecarAndMempoolTxs`. This function takes the following steps:

- Iterates over all `SidecarTx` in `sidecarTxs`. If all fit within maxima `maxBytes` and `maxGas`, then it adds all transactions to the `txs` array in order. If they do not fit, the function returns the transactions in `memplTxs`. This guarantees two things: atomicity for bundles and no maximum is exceeded (reaping from the mempool already checks maxima).

- Iterates over all mempool transactions. For each transaction, the function does two checks. It first checks that the transaction is not already in the `txs` array. This is to filter out duplicates. Second, it checks that by adding the transaction to the `txs` array, no maximum is exceeded. If any maximum is exceeded, then the function returns `txs`. Otherwise, it adds the transaction to `txs` and continues iterating.

```
function CombineSidecarAndMempoolTxs(
    memplTxs: ReapedTxs,
    sidecarTxs: ReapedTxs,
    maxBytes: int,
    maxGas: int
    ): Tx[] {

    runningGas = 0
    runningSize = 0
    txsMap = Map<TxKey, bool>
    txs = Tx[]

    for (let index in sidecarTxs.txs) {
        scTx = sidecarTxs.txs[index]
        dataSize = ComputeSizeTx(scTx)
        if ( maxBytes > -1 && runningSize + dataSize > maxBytes) {
            return memplTxs.txs
        }

        runningSize += dataSize

        if ( maxGas > -1 && runningGas + sidecarTxs.gasWanteds[index] > maxGas ) {
            return memplTxs.Txs
        }

        runningGas += sidecarTxs.gasWanteds[index]
        txs = txs.push(scTx)
        txKey = Key(tx)
        txsMap.set(txKey, true)
    }

    for (let index in memplTxs.txs) {
        memplTx = memplTxs.txs[index]
        // Filter out duplicates
        txKey = Key(tx)
        if txsMap.has(txKey) {
            continue
        }

        dataSize = ComputeSizeTx(memplTx)
        if ( maxBytes > -1 && runningSize + dataSize > maxBytes) {
            return txs
        }

        runningSize += dataSize

        if ( maxGas > -1 && runningGas + memplTxs.gasWanteds[index] > maxGas ) {
            return txs
        }
        runningGas += memplTxs.gasWanteds[index]
        txs = txs.push(memplTx)
    }

    return txs
}
```

## Properties Analysis

### Summary

| Property | Is it guaranteed? | Depends on Assumption #1 |
|---|---|---|
| No-disruption | ✅ | NO |
| Atomicity | ✅ | YES |
| Order | ✅ | YES |
| At-most-once | ✅ | YES |
| Priority | ❌ | YES |
| Optimal-reaping | ❌ | NO |
| Privacy | ✅ | YES |
| Accountability | ❓ | YES |

### Properties analysis

#### Property 1 [No-disruption] is guaranteed

No-disruption is guaranteed by the current design. This follows from the fact that the design implements a push-based model.

- Sentinels send bundles to proposers in the background, out of the consensus operation critical path.
- Block creators reap any complete bundle available at their local sidecar mempool at propose time, without engaging in any synchronous communication with sentinels or any other component.

The above guarantees that consensus is not delayed due to any bundle-related logic at the cost of weakening the trader guarantees. As argued below, the No-disruption property is in tension with the Priority property.

#### Property 2 [Atomicity] is guaranteed

Atomicity is guaranteed by the current design. The property is guaranteed if the array of transactions output by the function `CombineSidecarAndMempoolTxs` only includes complete bundles. This follows from three claims:

1. By Assumption 1, sentinels are trustworthy. This means that bundles are well-formed.
2. The function `ReapMaxTxs` only reaps complete bundles.
3. The function `CombineSidecarAndMempoolTxs` either includes all reaped bundles or none.

Proof Sketch: By (1) any complete bundle in a validator's sidecar mempool is well-formed: it satisfies atomicity. Then by (2), all reaped bundles satisfy atomicity. Finally by (3), all bundles included in a block satisfy atomicity, as required.

### Property 3 [Order] is guaranteed

Order is guaranteed by the design. The property is guaranteed if the array of transactions output by the function `CombineSidecarAndMempoolTxs` only includes complete bundles and the order defined by the array respects the order defined by the traders that submitted the bundles. The former follows from the Atomicity property. The latter follows from three claims:

1. By Assumption 1, sentinels are trustworthy. This means that bundles are well-formed.
2. The function `ReapMaxTxs` flattens bundles into an array of transactions. The explicit order of transactions in this array preserves the order within each bundle.
3. The function `CombineSidecarAndMempoolTxs` takes the array of transactions produced by `ReapMaxTxs` and produces a second array that respects the order defined by the former.

Proof Sketch: By (1) any complete bundle in a validator's sidecar mempool is well-formed: the inner transactions of a bundle are ordered according to the order defined by the bundle's trader. Then by (2), the array of transactions output by `ReapMaxTxs` respects the order defined by traders. Finally by (3), `CombineSidecarAndMempoolTxs` produces an array that respects the order defined by traders, as required.

### Property 4 [At-most-once] is guaranteed

At-most-once is guaranteed under the assumption that there are no duplicates in the Tendermint mempool. Given this assumption, the property follows from two claims:

1. By Assumption 1, sentinels are trustworthy. This means that for a given height, sentinels guarantee that a transaction is included in at most one bundle and at most once.
2. Before adding a mempool transaction to the final array of transactions, the function `CombineSidecarAndMempoolTxs` checks if the transaction already belongs to a reaped bundle. If that's the case, the mempool transaction is not included in the final array.

Proof Sketch: Claim 1 and the assumption that the Tendermint mempool has no duplicates guarantees that `memplTxs` and `sidecarTxs` set of transactions used as input of the function `CombineSidecarAndMempoolTxs` have not duplicates. By (2), follows that the merged array does not have duplicates as required.

Note that the caches implemented by both mempools do not prevent duplication because they have a maximum capacity.

> Cross-block non-duplication *is not guaranteed and it is up to the application to implement an application-specific replay protection mechanism with strong guarantees as part of the logic in* `CheckTx`.

### Property 5 [Priority] is not guaranteed

Priority is not guaranteed by the design. Fixed a chain and assume that the chain sentinel accepts two bundles `b1` and `b2` for a given height `h` such that the fee paid by the trader of `b1` is greater than the one paid by the trader of `b2`. By Priority either both are included in the block, none, or only `b1`. Under the assumption that messages can be dropped or reordered (Assumption 2), the protocol allows scenarios in which `b2` is included in the block and `b1` is not, even when sentinels follow the protocol (Assumption 1) and send bundle in increasing `bundleId` order.

For instance, consider the following scenario:

- The message containing the last transaction of `b1` is delayed and has not been received by the first proposer of height `h` by propose time.

- All other transactions for `b1` and `b2` have been received.
- When the proposer calls `ReapMaxTxs`, `b1` is not complete, so it will not be reaped. Nevertheless, `b2` is complete and will be reaped.
- If the block is committed, it will include `b2` and not `b1`, violating Priority.

The main reason why the protocol does not guarantee Priority is that `ReapMaxTxs` reaps any complete bundle, without caring whether bundles with smaller ids are guaranteed to be reaped. Interestingly, the property is weak enough to be easily guaranteed by the protocol without major changes: one simply needs to modify `ReapMaxTxs` to only reap a bundle with id `n` if all bundles with ids `<n` have been already successfully reaped.

> *Ideally, one would like to guarantee the stronger Strict-priority property: Priority with the constraint that all bundles that fit within the size and gas limits should be included in the block.*

Unfortunately, Strict-priority is in tension with No-disruption: it is very likely that to guarantee Strict-priority, the protocol would require synchronous communication with sentinels on propose time.

## Property 6 [Optimal-reaping] is not guaranteed

Optimal-reaping is not guaranteed by the protocol. There are scenarios in which bundles are not included in a block even though the proposer stores them entirely in its sidecar mempool at propose time.

The worst-case scenario is when the proposer has received all bundles on time and it does not include any in the block. This can happen if the total size or gas required by the set of bundles at the proposer by proposal time exceeds any limit. In that case, the protocol will only include the set of transactions reaped from the Tendermint's mempool (see function `CombineSidecarAndMempoolTxs`). This is to guarantee Atomicity: at this point, there is no bundle information, so excluding some transactions may compromise Atomicity. One simple way to solve this issue would be to make the function `ReapMaxTxs` check limits (similarly to what the function `ReapMaxBytesMaxGas` of the Tendermint's mempool does). This way one knows that the set of sidecar transactions used as an argument in `CombineSidecarAndMempoolTxs` won't exceed any limit.

There is a more subtle issue related to the sidecar's `maxBundleId` variable. The logic around `maxBundleId` implicitly assumes that sentinels and mev-validators are somehow synchronized. This is not true and may lead to a scenario in which a bundle with a high id is not reaped by the proposer even when it has been fully received by propose time:

- Assume that a mev-validator `val` is the first proposer of height `h`.
- Assume that the chain sentinel learns about the decision of height `h-1` much earlier than `val`.
- It is possible that `val` receives a bunch of bundles for height `h` before it learns the decision for `h-1`.
- When `val` learns the decision for `h-1`, it calls the `Update` function and sets `maxBundleId=0`.
- Assume that between the moment it calls `Update` and the propose time for height `h`, `val` does not receive any message from the sentinel.
- Then `val` will at most reap bundle 0 because the function `ReapMaxTxs` iterates from 0 to `maxBundleId`, which is 0 in this case.

A simple solution is to maintain a `maxBundleId` per bundle instead of one variable for all bundles. Since the sidecar is cleaned after committing a block, this should no be very costly.

## Property 7 [Privacy] is guaranteed

Privacy is guaranteed by the protocol. For a given height, the chain sentinel only propagates bundles to the proposer of the first round of consensus. This way the number of validators that have access to other traders' bundles and fees is minimized to one (plus the sentinel which is trusted), as required.

Guaranteeing Privacy comes with a cost: if agreement is not reached in the first round of consensus, no bundle will be included in the block, which has an impact on profit. Note nonetheless that this is not in tension with Priority: the property is weak enough to allow this type of scenarios. Nevertheless, Privacy would be in tension with Strict-priority, which disallows, among others, those scenarios in which no bundle is included in a block for a height for which the chain sentinel accepted bundles.

### Property 8 [Accountability] is guaranteed

This property could be guaranteed under the assumption that front-running and sandwich bundles can always be detected by sentinels.

> *NOTE: Out of the scope of this audit.*

## Findings

| Title | Issue |
|---|---|
| v1 Tendermint's mempool isn't safe | https://github.com/informalsystems/audit-skip/issues/10 |
| Failing consensus commit due to error updating mempools | https://github.com/informalsystems/audit-skip/issues/9 |
| Sidecar mempool is not locked on Commit() | https://github.com/informalsystems/audit-skip/issues/7 |
| Redundant bundle metadata in sidecar mempool | https://github.com/informalsystems/audit-skip/issues/5 |
| Inefficient code in ReapMaxTxs | https://github.com/informalsystems/audit-skip/issues/4 |
| Suboptimal reaping due to CombineSidecarAndMempoolTxs | https://github.com/informalsystems/audit-skip/issues/3 |
| Suboptimal reaping due to maxBundleId | https://github.com/informalsystems/audit-skip/issues/2 |
| Weak guarantees for traders | https://github.com/informalsystems/audit-skip/issues/1 |
| List of minor comments in diff PR | https://github.com/informalsystems/audit-skip/issues/6 |

## v1 Tendermint's mempool isn't safe

### Description

The CometBFT team has identified some issues in the v1 mempool implementation. It plans to deprecate it in v0.38.x as the use cases it enables overlap those enabled by PrepareProposal/ProcessProposal.

### Recommendation

We recommend not using the v1 implementation of the mempool.

### Status Update

Acknowledged.

## Failing consensus commit due to error updating mempools

### Description

The current implementation fails consensus commit if there is an error updating any of the mempools (this and this code). Interestingly, none of the `Update` methods of the mempools (this applies at least for v0 Tendermint's mempool) return an error: the methods always return nil.

### Problem Scenarios

At the moment, if commit fails, the error is logged but no action is taken. This is been marked as a bug though, found in the context of this issue in the CometBFT repo. This PR fixes this by crashing CometBFT when commit fails. This is because it is impossible to recover from some errors at commit that leaves the app in an undefined state.

### Recommendation

- We recommend removing those lines of code for now: it is dead code.
- In the future, we recommend failing commit only if the error is irreparable. A priori, it looks like failing commit because the sidecar mempool is not properly cleaned up is not one of those cases. Nevertheless, this must be analyzed case by case.
- Consider cherry-picking the fix to issue #490.
- If these lines of code are there to support v1 Tendermint's mempool, be aware that:
    - The CometBFT team has identified some issues in the v1 mempool implementation.
    - The CometBFT team plans to deprecate it in v0.38.x as the use cases it enables overlap those enabled by PrepareProposal/ProcessProposal

### Status Update

Removed dead code in skip-mev/mev-tendermint#126.

## Sidecar mempool is not locked on Commit()

### Description

On consensus `Commit`, Tendermint's mempool is locked to guarantee that all pending `CheckTx` calls are responded to and no new ones can begin. This is because `CheckTxState` (the state against which `CheckTx` runs) should be reset to the latest committed state at the end of every `Commit` (more here).

The current implementation does not lock the sidecar mempool on `Commit`.

### Problem Scenarios

Transactions are added to the sidecar mempool without running `CheckTx` on them, so the above is not a concern. Nevertheless, not locking allows for all kinds of data races. This is because `AddTx` and `Update` can be called concurrently and both are reading and updating a whole bunch of sidecar variables: `txs`, `txsMap`, `txsBytes`, `cache`, `maxBundleId`... This can lead to some non-ideal situations, i.e., the sidecar mempool stores transactions with `desiredHeight < sc.heightForFiringAuction`.
None of the non-ideal situations found seem harmful though. For instance, the above situation is not a problem because those transactions will be removed next time the sidecar mempool is cleaned on `Update`.

### Recommendation

It is generally recommended to get rid of data races https://go.dev/ref/mem. Thus, unless there is a specific reason to avoid locking the sidecar mempool (we do not think this will impact performance significantly), we recommend locking before calling `Update` to eliminate data races, which makes the code easier to understand, maintain and extend.

### Status Update

Fixed in skip-mev/mev-tendermint#126.

## Redundant bundle metadata in sidecar mempool

### Description

A Sidecar transaction include the following fields:

```
// MempoolTx is a transaction that successfully ran
type SidecarTx struct {
    DesiredHeight int64 // height that this tx wants to be included in
    BundleID      int64 // ordered id of bundle
    BundleOrder   int64 // order of tx within bundle
    BundleSize    int64 // total size of bundle

    GasWanted int64    // amount of gas this tx states it will require
    Tx        types.Tx // tx bytes

    // ids of peers who've sent us this tx (as a map for quick lookups).
    // senders: PeerID -> bool
    Senders sync.Map
}
```

### Problem Scenarios

`DesiredHeight`, `BundleID` and `BundleSize` are the same for all transactions within a bundle and it is already recorded for bundles. There is not need to maintain `BundleID` and `BundleSize`. Note that we still need `DesiredHeight` because it is used in `Update`.

### Recommendation

Redefine the `SidecarTx` struct to:

```
// MempoolTx is a transaction that successfully ran
type SidecarTx struct {
        DesiredHeight int64 // height that this tx wants to be included in
    BundleOrder   int64 // order of tx within bundle

    GasWanted int64    // amount of gas this tx states it will require
    Tx        types.Tx // tx bytes

    // ids of peers who've sent us this tx (as a map for quick lookups).
    // senders: PeerID -> bool
    Senders sync.Map
}
```

## Status Update

The Skip team is not going to address this for now because these fields, although redundant for now, make a `SidecarTx` fully identifiable w.r.t. its parent bundle, which may be useful in some future changes and isn't a major efficiency issue right now.

## Inefficient code in ReapMaxTxs

### Description

A proposer only reaps complete bundles in `ReapMaxTxs`. The issue is that even when the function detects early that the bundle is not complete, it still iterates over all possible inner transaction ids (`0` to `bundle.EnforcedSize-1`) and completeness is only checked afterwards. This is the piece of code:

```
for bundleOrderIter := 0; bundleOrderIter < int(bundle.EnforcedSize);
bundleOrderIter++ {
    bundleOrderIter := int64(bundleOrderIter)
    if scTx, ok := bundleOrderedTxsMap.Load(bundleOrderIter); ok {
        // loading as sidecar tx, but casting to MempoolTx to return
        cTx := scTx.(*SidecarTx)
        innerTxs = append(innerTxs, scTx)
    } else {
        // can't find tx at this bundleOrder for this bundleID
        sc.logger.Info(
            "In reap: skipping bundle, don't have a tx for bundle",
            "bundleID", bundleIDIter,
            "bundleOrder", bundleOrderIter,
            "height", sc.heightForFiringAuction,
        )
    }
}
// check to see if we have the right number of transactions for the bundle, comparing
to the enforced size
if bundle.EnforcedSize == int64(len(innerTxs)) {
...
```

### Recommendation

Even if this is not critical (bundle size is expected to be small as far as we understand), this is unnecessarily inefficient.

### Status Update

Fixed with a break statement in skip-mev/mev-tendermint#131.

## Suboptimal reaping due to CombineSidecarAndMempoolTxs

### Description

The protocol should guarantee that a correct block proposer reaps as many bundles from the sidecar mempool as possible to maximize profit. This is formalized in the spec by the Optimal-reaping property.

### Problem Scenarios

A proposer combines sidecar and mempool transactions via the `CombineSidecarAndMempoolTxs` function. If the set of sidecar transactions exceeds any maximum (total size or gas), the function returns the set of mempool transactions. This implies that the block won't include any bundle, even if these were all received on time. This is suboptimal.

### Recommendation

We understand that this is done to guarantee the atomicity of bundles: at this point there is not bundle information, so excluding some transactions may compromise atomicity. Nevertheless, we think this could be easily fixed. One simple way to solve this issue would be to make the function `ReapMaxTxs` check limits (similarly to what the function `ReapMaxBytesMaxGas` of the Tendermint's mempool does). This way one knows that the set of sidecar transaction used as argument in `CombineSidecarAndMempoolTxs` won't exceed any limit.

### Status Update

The Skip team has decided not to address this issue. The implementation assumes that the set of bundles sent by the chain sentinel to the block proposer of a given height never exceeds any of the maxima. If the sentinel sends a set of bundles that exceeds either of the limits, the implementation interprets this as a sign of a malicious or misbehaving sentinel, and thus falls back to tendermint’s mempool.

## Suboptimal reaping due to maxBundleId

### Description

The protocol should guarantee that a correct block proposer reaps as many bundles from the sidecar mempool as possible to maximize profit. This is formalized in the spec by the Optimal-reaping property.

### Problem Scenarios

The logic around `maxBundleId` implicitly assumes that sentinels and mev-validators are somehow synchronized. This is not true and may lead to a scenario in which bundles are not reaped by the proposer even when they have been fully received by propose time. An example execution is the following:

- Assume that a mev-validator `val` is the first proposer of height `h`.
- Assume that the chain sentinel learns about the decision of height `h-1` much earlier than `val`.
- It is possible that `val` receives a bunch of bundles for height `h` before it learns the decision for `h-1`.
- When `val` learns the decision for `h-1`, it calls the `Update` function and sets `maxBundleId=0`.
- Assume that between the moment it calls `Update` and the propose time for height `h`, `val` does not receive any message from the sentinel.
- Then `val` will at most reap bundle 0 because the function `ReapMaxTxs` iterates from 0 to `maxBundleId`, which is 0 in this case.

### Recommendation

A simple solution is to maintain a `maxBundleId` per bundle instead of one variable for all bundles. Since the sidecar is cleaned after committing a block, this should not be very costly.

### Status Update

Addressed in skip-mev/mev-tendermint#132 by keeping a maxBundleID per height.

## Weak guarantees for traders

### Description

The sentinel of a chain opens an auction for a block and traders submit bundles. A desirable property for traders would be that if a trader's bundle was accepted in the auction but did not make it into the committed block, then no bundle with a lower associated fee made it. This is formalized in the spec by the Priority property:

Assume that for a given auction the sentinel has received two bundles `b1` and `b2` such that the trader that submitted `b1` is willing to pay a higher fee. By this property, the protocol should guarantee that if only one bundle is included in the auctioned block, then it has to be `b1` : `b1` has priority over `b2` . The property also allows both bundles to be included or none.

### Problem Scenarios

The current implementation does not guarantee the above property. Fixed a chain and assume that the chain sentinel accepts two bundles `b1` and `b2` for a given height *h* such that the fee paid by the trader of `b1` is greater than the one paid by the trader of `b2` . By Priority either both are included in the block, none, or only `b1` . Under the assumption that messages can be dropped or reordered, the protocol allows scenarios in which `b2` is included in the block and `b1` is not, even when sentinels follow the protocol (are trustworthy) and send bundles in increasing `bundleId` order.

Consider now the following scenario:

- The message containing the last transaction of `b1` is delayed and has not been received by the first proposer of height `h` by propose time.
- All other transactions for `b1` and `b2` have been received.
- When the proposer calls `ReapMaxTxs` , `b1` is not complete, so it will not be reaped. Nevertheless, `b1` is complete and will be reaped.
- If the block is committed, it will include `b2` and not `b1` , violating Priority.

### Recommendation

If the Skip team believes that strengthening traders' guarantees is important, we suggest the following simple change.

The main reason why the protocol does not guarantee Priority is that `ReapMaxTxs` reaps any complete bundle, without caring whether bundles with smaller ids are guaranteed to be reaped. Interestingly, the property is weak enough to be easily guaranteed by the protocol without major changes: one simply needs to modify `ReapMaxTxs` to only reap a bundle with id `n` if all bundles with ids `<n` have been already successfully reaped.

### Status Update

The Skip team has decided not to address the issue for now:

- Currently, there's only one winning bundle per height.
- Additionally, when there are going to be multiple winners in the future, it is unclear whether they want to guarantee Priority.

## List of minor comments in diff PR

### Description

#### Related to the sidecar mempool

- You may want to make the size of the cache configurable (https://github.com/angbrav/mev-tendermint/pull/1#discussion_r1121578073)
- Inaccurate login info (https://github.com/angbrav/mev-tendermint/pull/1#discussion_r1121903932)
- `updateMtx` is never acquired for writing, so acquiring for reading has no effect (https://github.com/angbrav/mev-tendermint/pull/1#discussion_r1121944837)
- Wrong comparison operator when checking if a bundle is complete on `AddTx` (https://github.com/angbrav/mev-tendermint/pull/1#discussion_r1121970019)
- Wrong comparison operator when updating `maxBundleId` (https://github.com/angbrav/mev-tendermint/pull/1#discussion_r1122004863)
- Sidecar's `height` field is never read (https://github.com/angbrav/mev-tendermint/pull/1#discussion_r1122042305)
- Misleading comment (https://github.com/angbrav/mev-tendermint/pull/1#discussion_r1122957463)
- Misleading comment (https://github.com/angbrav/mev-tendermint/pull/1#discussion_r1124712831)
- Bundle's `gasWanted` is not used (https://github.com/angbrav/mev-tendermint/pull/1#discussion_r1123329374)
- Abuse of anonymous function (https://github.com/angbrav/mev-tendermint/pull/1/files#r1129746461)

#### Related to the p2p layer

- It seems that mevMetrics.SentinelConnected is currently set to 0 in case of removal NOT succeeded. This part of the code should be moved to the if branch - when removal of the sentinel is successful. (https://github.com/angbrav/mev-tendermint/pull/1#discussion_r1122744503)
- Logging this message in case of pure optimization of gossip could be misleading. (V0: https://github.com/angbrav/mev-tendermint/pull/1#discussion_r1127809651, V1: https://github.com/angbrav/mev-tendermint/pull/1#discussion_r1127824207)

### Status Update

All addressed with skip-mev/mev-tendermint#131 and skip-mev/mev-tendermint#126.