Ren

Code Assessment

Hyperdrive

PwC Switzerland

May 26, 2020

Strictly private and confidential

Contents

Dappbase Pte. Ltd.
#19-08 Prudential Tower, 30 Cecil Street
049712 Singapore

1 Executive Summary

Dear Loong,

First and foremost we would like to thank Ren for giving us the opportunity to assess the current state of their Hyperdrive system. This document outlines the findings, limitations, and methodology of our assessment.

PwC Switzerland was tasked by Ren to perform a code assessment of the Hyperdrive project. Hyperdrive is a novel consensus engine that combines tendermint-style consensus with two new features: fast-forwarding and rebasing. A consensus engine is a critical component in building a decentralized system. However, further components are needed. In the case of Hyperdrive the following components need to be added: networking, data storage, transaction structure, block structure, and the respective validation functions. Hence, this report cannot determine the overall security of a system built upon Hyperdrive as it is only concerned with Hyperdrive itself.

Furthermore, the usability and stability of the Hyperdrive features depend on the way that the previously mentioned components are implemented and integrated. The fast-forwarding feature allows a special operation that most blockchains do not support: jumping ahead without the need to verify intermediate blocks. This also implies that a local node does not necessarily guarantee traceability as it might not hold all blocks. Furthermore, fast-forwarding requires a state transfer which implies that this transfer needs to be feasible size-wise.

During the assessment several findings were discovered. Almost all were resolved and can be seen in the Resolved Findings section. Only one low-severity finding, which is common to this type of consensus protocols, was not addressed. Due to the complexity of consensus algorithms in general, there is an inherent risk that more issues exist in the implementation. Previous examples of consensus mistakes being discovered after more than a decade [2], highlight how difficult they are to secure given their complex states. We hence recommend the use of a complex test suite, e.g. Twins [2].

We hope that this assessment provides more insight into the current implementation and provides valuable findings. We are happy to receive questions and feedback to improve our service and are highly committed to further support your project.

Yours sincerely,

PricewaterhouseCoopers AG

A. Eschbach

Andreas Eschbach

[illegible]

Hubert Ritzdorf

2 Assessment Overview

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

2.1 Scope

The general scope of the assessment is set out in our engagement letter with Ren dated March 10, 2020. The assessment was performed on the source code files inside the Hyperdrive repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

Date	Commit Hash	Note
March 11, 2020	81f40ed84800a36d457fe21b22758a022e7bc79a	First Version
March 25, 2020	f12dec56d5e83d8e0ea2cc44316e8b8f9c10953d	After First Report
May 4, 2020	edcc757304b2375c6ed6674c74c06edfebd2aff3	After Second Report
May 21, 2020	1357f1878365b9e3ad7d1957b41037652dc496e5	After Further Comments

2.1.1 Excluded from scope

The surge library for marshalling and unmarshalling is excluded from the scope as it was added during the process of the engagement.

2.2 System Overview

The hyperdrive package implements the consensus algorithm described in the paper *The latest gossip on BFT consensus* [1]. Some additional features, namely fast-forwarding and rebasing have been added.

Hyperdrive is designed in order to be agnostic towards the structure of transactions or blocks. Therefore anyone may use this consensus package. A core use case is its usage in the `RenVM` implementation.

The hyperdrive package interacts with other components. In particular it relies on the existence of a peer-to-peer network with broadcast capabilities and a storage interface for the state.

The implemented consensus algorithm has the fairly well-known properties of byzantine fault tolerance. The nodes participating in the current consensus are called signatories. Based on the existence of `3f+1` signatories with a maximum of `f` malicious or faulty nodes, consensus on abstract data can be reached. Participating nodes only need to know the latest block to be able to participate in the consensus for the next block.

2.3 Terminology

In the following part we introduce some terminology that is relevant for this report.

2.3.1 Blocks

Blockchains using the hyperdrive consensus implementation know 3 types of blocks:

`standard`
: Standard blocks are the most common blocks in the system, containing application specific data like transactions.

`rebase`
: Rebase blocks initiate the change of signatories that govern the consensus algorithm.

`base`
: Base blocks finalize the change of signatories.

Important terms in connection with a block are `height` and `round`. The height is the number of the block. The blockchain starts at block height 0, the first block is a base block containing the initial signatories. The next block is at height 1. At each height, there may be one or more rounds needed until consensus has been reached. At each new block height, round 0 starts to find consensus. In case no consensus has been reached (this can be the case if more than `2f` precommits for `nil` have been received at this height and round), the scheduled proposer failed to propose the next block.

A new round at this height is invoked where the next proposer is tasked to propose a new block. This continues until consensus at this height has been found, then the consensus for the next height starts again at round 0.

2.3.2 Messages

To reach consensus about the next block, different types of messages are exchanged over the network:

`Propose`
: The scheduled proposer at the current height & round crafts a new propose message. This message is a proposal for the next block.

`Prevote`
: Signatories react to propose message for the current height & round by sending a prevote message.

`Precommit`
: Upon having received enough prevotes for a certain block proposal, signatories precommit for this block if they find the block to be valid.

All messages are for a specific blockhash (or invalid hash) and a specific height & round.

Additionally, `Resync` messages have been introduced. These messages are used to query others for previously sent messages after a node recovers from a crash.

`prevote nil` and `precommit nil` are the terms used to vote against a proposal at a certain height or round. These are simply messages for the invalid blockhash and are used if the node does not agree with the proposal or if it has not received enough messages before the timeout elapsed.

`fast-forward`: Nodes that have fallen out-of-sync are able to fast-forward to the latest block by receiving the latest block the consensus has agreed on, endorsed by sufficient (`2f+1`) precommits. These information are included in the proposal message. Only the latest block is required in order to participate in the consensus for the next height. Note that in case of fast-forwarding, if required, the missing blocks have to be synced separately.

`rebase`: A rebase is the process of changing the signatories of the consensus. A rebase block including the new set of signatories is proposed and if accepted immediately followed by a base block enabling the new signatories.

`Replica`: A Replica represents one process of the hyperdrive consensus algorithm, a replicated state machine. Each replica is bound to a specific shard. It takes part in the hyperdrive consensus and sends and receives messages.

2.3.3 Roles

`Signatories`

Set of chosen signatories taking part in the consensus process.

`Proposer`

The proposer is the signatory chosen by the scheduler to propose the next block at a specific height and round.

`Scheduler`

Algorithm responsible to return the signatory to propose a block at a specific height & round.

`Observer`

An observer may be called upon certain events:

- `DidCommitBlock()`
- `DidReceiveSufficientNilPrevotes()`

Additionally, a boolean `IsSignatory` indicates the current state of the observer

`Validator`

A validator exposes `IsBlockValid` to hyperdrive. This function is used to validate a block. **It's of paramount importance that this function is correctly implemented.** This is because Hyperdrive trusts this function totally.

`shardRebaser`

Each replica has it's `shardRebaser` tasked to handle a rebase

`Broadcaster`

The Broadcaster is responsible to distribute the messages of these nodes throughout the network. The implementation is not defined by the hyperdrive package and may vary. **For the functional correctness of the hyperdrive algorithm it is important that messages crafted and broadcasted by this very node also arrive and are processed at this node.** In particular a crafted and broadcasted proposal by a node must also arrive at the sending node to trigger a prevote message. Similarly for other message types, these need to arrive at their originating node as well to trigger important functionality like counting for the `2f+1` prevotes. Otherwise, in the case of exactly `f` adversaries (the maximum) only `2f` prevotes would be emitted and arrive at the nodes - insufficient to pass the barrier of `2f+1` to generate a precommit.

2.3.4 Overview of the consensus process

To be able to participate in the consensus about the next block a node only needs to know the latest valid block and the current signatories.

Upon the start of a new round of consensus, the proposer (the scheduled signatory) crafts and broadcasts a new block proposal. All other nodes invoke a timeout and - in case - no proposal arrives, they will automatically prevote `nil`. This guarantees that the consensus is not blocked upon a single missing proposal.

Upon processing a proposal messages for the current height & round, signatories either accept it and endorse it by prevoting for this blockhash or prevote nil to oppose this block. Next, nodes collect `2f+1` prevotes in order to send out a precommit.

Once `2f+1` precommit messages for a blockhash at a certain height & round are reached, the proposed block is accepted and a new round at the next height is started once a node has received enough precommits.

Messages may arrive out-of-order or fail to arrive completely due to various reasons such as network congestion or network partition. All received messages are stored in an Inbox and are counted. This ensures that all relevant messages are counted when a node decides on its actions, even in case these messages arrive out-of-order.

2.3.5 Trust Model

This Byzantine Fault Tolerant consensus algorithm handles up to `f` out of `3f+1` malicious / offline signatories. A particular signatory is not necessarily trusted. The code using the hyperdrive package is assumed to be non-malicious. As a special feature, some of the `2f+1` honest nodes, might temporarily crash, but the consensus can recover using the `Resync` messages.

3 Limitations and use of report

3.1 Inherent limitations

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

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

3.2 Restriction of use and purpose of the report

Our report is intended solely for Ren for use in connection with the purpose as described in the preceding paragraph. Our report should not be distributed to or used by parties other than Ren or used for any other purpose. We do not, in giving our opinion, accept or assume responsibility or liability for any other purpose or to any other parties to whom our report is shown or into whose hands it may come.

4 Terminology

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

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

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

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

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

5 Findings

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

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

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

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	1

- Consensus is not resilient due to single point of failure Risk Accepted

5.1 Consensus is not resilient due to single point of failure Security Low Risk Accepted

For the network to move on, the consensus algorithm needs to agree on new blocks. In each height and round however there is a single point of failure: the scheduled proposer. If the scheduled proposer fails to propose, the consensus moves on to the next round at this height, but does not agree on the next block. At the next round, the next scheduled proposer is tasked to propose.

As the scheduler must be known to all participants, an attacker is able to predict the sequence of the next proposer. Assuming signatories run a replica and participate in the network (their IP address is known) an Attacker could simply DoS the next scheduler and prevent him from successfully broadcasting his propose. After the timeout, the attacker would DoS the next scheduled proposer. All in all, if successful, an attacker may be able to stop the network (or at least degrade the network) with a relatively low overhead - he just needs to DoS one node at a time.

While this is a common issue in leader-based consensus protocols, we still want to point it out, as most blockchain consensus protocols are not leader-based and hence to do not suffer from this. Essentially, when using this protocol nodes should implement some stability mechanisms.

Risk Accepted: The hyperdrive team states: *We will consider ways in which this can be fixed in future versions. It is not immediately obvious how this can be resolved in any PBFT-based algorithm without reasonable amounts of additional complexity. Until then, we are reliant on three mechanisms to mitigate this issue: rate-limiting implemented by the user of Hyperdrive, cloud provider protections against network-level attacks, and increasing timeouts as rounds progress (requiring the attacker to maintain attacks for longer and longer)*

6 Resolved Findings

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

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

Critical-Severity Findings	0

High-Severity Findings	6

- Cross-Shard Fast Forward Code Corrected
- Findings related to `Inbox` structure Code Corrected
- Incomplete signature of propose messages Code Corrected
- Scheduler has incorrect signatories during Rebase Code Corrected
- Wrong base block signatories used for fast-forwarding Code Corrected
- `f` not updated when signatories change Code Corrected

Medium-Severity Findings	9

- Consensus stuck due to dropped Proposal Code Corrected
- Fast forwarding across rebase needs to update signatories Code Corrected
- Fast-forwarding does not notify the observer Code Corrected
- Incorrect precommits interfere with fast-forward functionality Code Corrected
- Malformed input data causes panic Code Corrected
- No catch up after missing base block Specification Changed
- Precommits may be blocked if proposal arrives last Code Corrected
- Shard not included in signature Code Corrected
- Single-time trigger can be triggered twice Code Corrected

Low-Severity Findings	9

- Checking the `BaseHash` as part of block validity Code Corrected
- Consistent handling for commits Code Corrected
- Different Blocks can have same timestamps Code Corrected
- Inconsistent handling in Message Queue Code Corrected
- No round check performed Specification Changed
- One Signatory only Code Corrected
- Possible Blockhash Collision Code Corrected
- Signatory can DoS using fast-forward Code Corrected
- TODO in Code Code Corrected

6.1 Cross-Shard Fast Forward Security High Code Corrected

In case signatories from one shard, become signatories of another shard (either simultaneously or later), an attacker can reuse their signatures to perform an incorrect fast-forward. The attack is explained in an example below:

Setting:

- Shard 1 has significantly higher block numbers, e.g. current block number is 1,000
- Shard 2 has a current block number of 100
- The current signatories of shard 2 were signatories of shard 1 during blocks 1 - 500

Attack:

- A malicious signatory of shard 2 creates the following proposal: - For a height when it would be scheduled - So that there would not be a rebase block in between - Example: proposal for block at height 151 - Include a `latestCommit` for block 150 of shard 1 - The proposed block correctly builds on the one referenced in `latestCommit` - It is signed for shard 2
- This proposal is received by the other signatories: - They check that the message has the right shard 2, which is correct - They check that the signatory is correct, which is true - They handle the propose message - Check that the attacker is scheduled to propose a block and insert it - They execute `syncLatestCommit` - The block is from the future and comes with the necessary signatures - The 2f+1 precommits are all for this block - All signatories are valid signatories - `syncLatestCommit` fast-forwards to 150

Summary:

- The attack doesn't necessarily require that signatories are active in parallel, they could also switch shards over time
- The overlap of signatories needs to be at least 2f+1
- These signatories do not have to be malicious, their signatures are simply reused by a single malicious signatory

---

Code Corrected:

The shard information is now part of each message, including precommits. When checking precommits before fast-forwarding each precommit is checked whether it belongs to the right shard. Hence, the finding is resolved.

6.2 Findings related to `Inbox` structure Security Design High Code Corrected

Hyperdrive currently processes and inserts all received messages into an inbox, given they are signed by a current signatory included in the current base block. This is done regardless of the block height of the message.

This method not only inserts the message but also returns information about how many messages of this height and round have already been seen.

```
func (inbox *Inbox) Insert(message Message) (n int, firstTime, firstTimeExceedingF, firstTimeExceeding2F, firstTimeExceeding2FOnBlockHash bool)
```

A comment states:

> "This method is used extensively for tracking the different conditions under which the state machine is allowed to transition between various states. Its correctness is fundamental to the correctness of the overall implementation."

The current implementation has several issues, summarized these are:

1. Outdated messages are inserted & processed - which is unnecessary
2. Silent overwriting of messages
3. Inserting messages for the future

Each of these three issues is described in detail below. Another separate issue addresses how the `firstTimeExceeding2FOnBlockhash` marker can return `true` multiple times for the same height & round.

1. Currently, all received messages are processed by the `handleXXX()` functions **even when the node knows that these are outdated**. This is a waste of cpu-time and may open up an attack surface for denial-of-service attacks. Note that each time a new block is confirmed (meaning that enough precommits have been received) all messages of previous heights are discarded. Nevertheless, if a new outdated message arrives again, it is inserted & processed again only to be dropped at the next confirmed block.

   Once consensus has been reached on a specific block height, messages concerning this block height or below are irrelevant and should not be processed anymore. The code features no such checks.

   A malicious broadcaster in the network may attempt to spam nodes by replaying outdated messages. Ideally, outdated messages are dropped as early as possible, e.g. even before verifying the signature.

2. Under ideal conditions, every signatory sends one (correct) message for each height, round and type which is received by everyone. Under real circumstances however, this is not the case. Signatories may act malicious, either intentionally or when their private key has been compromised. BFT guarantees the proper functionality of the network up to f malicious/unavailable nodes.

   Upon receiving a message, the following happens:

   A. The previous length of the inbox is read

      Using `_, ok := inbox.messages[height][round][signatory]` it is checked if an entry already exists. The value is discarded, `ok` is a boolean indicating if the entry in the map is already existing or not. The existence of an entry implies a previous message of this type from this signatory at this height and to this receiver.

   B. No matter the result, the entry is updated:

      ```
      inbox.messages[height][round][signatory] = message
      ```

      The message is stored in the map

```
inbox.messages[height][round][signatory]
```

C. In case a message was already present, it is overwritten. There is no check if it is overwritten by the very same or a different message of this signatory. Overwriting it with the very same message is unnecessary and detecting two different messages of the same signatory at the very same height and round should be followed up appropriately.

D. Next the new length of the map is read and the following behavior occurs:

```
if !ok {
        nOnBlockHash = inbox.QueryByHeightRoundBlockHash(height, round, message.BlockHash())
}
```

This means, only if there was no entry in `inbox.messages[height][round][signatory]`, a check is performed how many messages for the same blockhash have been sent by replicas already.

So in case this was a repeated message, this is not triggered. However with the next message of another signer, the newly overwritten message is included in the counting.

Executing `Insert` again for an identical message which has already been processed before should not change the system state and all the `firstTimeX` values should be false. (There is a separate issue which shows this is not necessarily the case for `firstTimeExceeding2FOnBlockHash`.) This is just additional computation overhead which could be avoided. Note that such messages may be replayed / spammed by anyone, thus the computational overhead upon handling a second, identical message must be minimal.

As a consequence of this overwriting of messages, the description of function `QueryMessagesByHeightRound` which states

"returns all unique messages that have been received"

is not totally true. It just returns all currently stored messages of all signatories at this height and round. More messages may have been received by a signatory, but have been overwritten in the meantime.

3. Inserting messages for **future heights and rounds** is problematic, as there may be rebase and base blocks in between the current height and the future height. The rebase procedure might change the set of signatories.

The `Verify()` function of `message.go` only checks if the signatory of the message is currently a member of the signatories listed in the latest base block, not if the signatory is a actual member of the actual signatories at the block height of the message. Nevertheless `handlePropose()`, `handlePrevote()` and `handlePrecommit()` just enter these message into the Inbox. Thus these messages influence the calculations later on (e.g. when the total length of the inbox is considered).

Currently, the signatory is checked at the time the message arrives against the set of currently active signatories irregardless of the height of the message. This allows a current signatory to insert a message for future heights where he may not be a valid signatory anymore.

While it's considered impossible to predict a valid blockhash for a future block, crafting messages using `InvalidHash` interfering with precommit/prevote `nil` counting is certainly doable. Furthermore, in case of colluding signatories, they may be able to insert enough messages e.g. precommitting for a non-existing blockhash, which may become very problematic:

Consider the following scenario. A malicious signatory *S* acts normally, but signs `precommit` messages for a blockhash *BH* at a future height *H*. Over time, some change of signatories, so-called rebasing happen. Each time *S* changes it's private key and hence can submit multiple such `precommit` messages for *BH* at height *H*. Once *H* is reached, it just needs to send a single message for *BH* and there might be 2f+1 signatures. Hence, a single signatory can determine a block on its own.

While these future messages could be legitimate messages, the node cannot know this at this point. Such messages may be queued and only processed once the node is at this specific height.

Code corrected: The mentioned issues have been addressed by changes in the code.

1. Outdated messages are inserted & processed: Outdated messages are no longer processed. `HandleMessage()` of `replica.go` now drops outdated messages immediately.

2. Silent overwriting of messages: The `Insert` Method of `Inbox` no longer allows overwritting of messages. In case of a second messages at the very same `height` and `round` by the same `Signatory`, `previousN, false, false, false, false` will now be returned.

3. Inserting messages for the future: Before processing messages from the queue a check now ensures that no base block has been missed.

6.3 Incomplete signature of propose messages

Security High Code Corrected

To ensure the integrity of a message, the hash of the message is signed. However the hash of the propose message does not include all fields. Missing fields can be manipulated despite the message having been signed. The definition in `message.go` shows that the `sha256.Sum256()` is calculated over the `String()` function of the message type (Propose, Prevote, or Precommit).

The propose messages are defined as follows:

```go
type Propose struct {
        signatory  id.Signatory
        sig        id.Signature
        height     block.Height
        round      block.Round
        block      block.Block
        validRound block.Round

        latestCommit LatestCommit
}
```

However, the hash is only done over the message which is constructed like this:

```go
fmt.Sprintf("Propose(Height=%v,Round=%v,BlockHash=%v,ValidRound=%v)",
            propose.Height(), propose.Round(), propose.BlockHash(), propose.ValidRound())
```

The struct entry `LatestCommit` is missing. An attacker may intercept a `Propose` message and change the content of `LatestCommit`. Combined with the Denial-of-Service issues exploiting `syncLatestCommit`, anyone may be able to launch Denial-of-Service attacks against nodes. Note that all fields except `signatory` and `sig` (which are part of the verification process) need to be included in the hash.

Code corrected: To fix issue *Shard not included in signature* the updated code introduces an additional signature for messages. The broadcaster now signs the hash of the marshalled representation of the message struct.

```
type Message struct {
        Message   process.Message
        Shard     Shard
        Signature id.Signature
}
```

This includes the whole marshalled representation of the included `process.Message` which resolves the issue mentioned above. However, each message now features two signatures.

6.4 Scheduler has incorrect signatories during Rebase Correctness High Code Corrected

When a rebase starts, the `Rebase` function is called for the `Replica`. This function also triggers the `scheduler` to rebase. The scheduler immediately performs the following operation:

```
func (rr *roundRobin) Rebase(signatories id.Signatories) {
    rr.signatoriesMu.Lock()
    defer rr.signatoriesMu.Unlock()

    // Copy signatories into the scheduler to avoid manipulation of the slice,
    // external to the scheduler, from affecting the scheduler.
    rr.signatories = make(id.Signatories, len(signatories))
    copy(rr.signatories, signatories)
```

This immediately replaces the signatories. Hence, any subsequent calls to the `scheduler` will return one of the new signatories. This can lead to deadlock situations as the new signatories cannot propose the rebase block, but their proposal might be expected according to the `scheduler`.

Code Corrected:

In the updated code, the `shardRebaser` triggers the scheduler's update only once it has received the notification that the `Base` has been committed. Hence, the set of signatories is updated at the right time.

6.5 Wrong base block signatories used for fast-forwarding Security Design High Code Corrected

The function `syncLatestCommit()` inside `process.go` processes the `LatestCommit` struct included in a proposal in order to allow fast forwarding. To ensure that the included precommits are signed by valid signatories, the code compares the used signatories to the signatories in the base block. Instead of loading the latest base block however, block 0 (the genesis base block), is always loaded and its signatories are extracted.

```
baseBlock, ok := p.blockchain.BlockAtHeight(0)
```

The signatories may have been updated during a rebase. The function `syncLatestCommit` however uses the original set of signatories defined in block 0, instead of the signatories defined in the latest base block. Hence the original signatories, which may have been replaced can still sign precommits included in the `latestCommit` struct and thereby may make out-of-sync nodes fast-forward to a potentially malicious block.

Code corrected: The code was changed to:

```
baseBlock := p.blockchain.LatestBaseBlock()
```

For related issues regarding the awareness of the latest base block further code changes were implemented in https://github.com/renproject/hyperdrive/pull/78.

6.6 `f` not updated when signatories change

Design High Code Corrected

Creating a new replica starts a new process. A process is initialized with a `state` which includes `Inboxes` for the messages. Upon creating an Inbox the parameter `f` of the inbox must be set. Starting a new instance of a replica creates a new process and for the state argument the following is passed: `process.DefaultState((len(latestBase.Header().Signatories())-1)/3)`. `f` is set to `len(latestBase.Header().Signatories())-1)/3` for each inbox.

During rebasing the signatories can change. Currently this is implemented as follow:

```
func (replica *Replica) Rebase(sigs id.Signatories) {
    if len(sigs)%3 != 1 || len(sigs) < 4 {
        panic(fmt.Errorf("invariant violation: number of nodes needs to be 3f +1, got %v", len(sigs)))
    }
    replica.scheduler.Rebase(sigs)
    replica.rebaser.rebase(sigs)
}
```

While it is ensured that the new length of the signatories fulfills the `3f+1` condition, the `f` of the `Inbox` is not updated. Thus, if the length of the signatories changes, the `firstTimeXXX` of the `Inbox` are calculated incorrectly using the unchanged, old value for `f`. The conditions under which the state machine is allowed to transition between various states won't be correctly triggered anymore.

Code corrected:

The `shardRebaser` now ensures that the length of the new signatories is equal to the length of the previous signatories, this mitigates the problem described above.

6.7 Consensus stuck due to dropped Proposal

Security Medium Code Corrected

The following messages are exchanged:

1. **Proposal, Height n, Type: Base**
    - This message will get a delayed delivery to f correct nodes
    - The other 2f+1 nodes receive it immediately
2. **Prevote, Height n**
    - Succeeds with 2f+1 nodes
3. **Precommit, Height n**
    - Succeeds with 2f+1 nodes
4. **Proposal, Height n+1, Type: Standard**
    - Thrown away by the f nodes that haven't received message 1, because they are missing a base block
    - Other 2f+1 nodes send prevotes
5. **Prevote, Height n+1**
    - Succeeds with 2f+1 nodes
6. **Precommit, Height n+1**
    - Succeeds with 2f+1 nodes
    - Right after sending the f faulty nodes **fail**
7. **Proposal, Height n+2, Type: Standard**
    - Thrown away by the f nodes that haven't received message 1, because they are missing a base block
8. **Now Message 1 reaches the f correct nodes**
    - They advance to height n+1 and wait for a Proposal
    - After the timeout, they send Prevote nil, but never receive 2f+1 prevotes

9. **Prevote, Height n+2**

    - Sent by the f+1 nodes that are up-to-date and received message 7
    - They never receive 2f+1 prevotes

The f nodes are stuck at height n+1 and f+1 nodes are stuck at height n+2. Hence, they never advance.

This example also works if only one node instead of f nodes gets a delayed delivery of message 1.

Code corrected: After a new base block has been committed a `Resync` message is sent out to ensure a synchronization of lost messages.

6.8 Fast forwarding across rebase needs to update signatories Design Medium Code Corrected

If fast forwarding happens over a across a set of rebase and base blocks the code using the Hyperdrive package must ensure that the signatories are updated accordingly. Considering only the currently implemented Hyperdrive code, this is not handled. Hence, the node performing the fast forwarding would attempt to proceed with the outdated signatories of the last base block it is aware of.

Code corrected:

Fast forwarding happens only with `Propose` messages, upon receiving a `Propose` message for a future height the updated code now checks for missed base blocks and only handles this `Propose` message if no base block has been missed.

```
// If the Propose is at a future height, then we need to make sure
// that no base blocks have been missed. Otherwise, reject the
// Propose, and wait until the appropriate one has been seen.
baseBlockHash := replica.blockStorage.LatestBaseBlock(m.Shard).Hash()
blockHash := m.Message.BlockHash()
numMissingBaseBlocks := replica.rebaser.blockIterator.BaseBlocksInRange(baseBlockHash, blockHash)
if numMissingBaseBlocks == 0 {
        // If we have missed a base block, we drop the Propose. The
        // Propose that justifies the next base block will eventually be
        // seen by this Replica and we can begin accepting Proposes from
        // the new base.

        // In this condition, we haven't missed any base blocks, so we
        // can proceed as usual.
        replica.p.HandleMessage(m.Message)
}
```

It is essential that `BaseBlocksInRange()` works as intended.

Dropping the `Propose` message in case a base block has been missed might be problematic, as such a message might still be legitimate. Messages might only arrive once and in this case, the message would never be processed.

6.9 Fast-forwarding does not notify the observer Correctness Medium Code Corrected

After seeing `2f+1` precommits for a block, this block is inserted using `InsertBlockAtHeight()` and - if available - the observer is notified using `DidCommitBlock()`.

The observer is the `Rebaser`, `DidCommitBlock` does the following depending on the type of the block:

```
case block.Standard:
case block.Rebase:
        rebaser.expectedKind = block.Base
        rebaser.expectedRebaseSigs = committedBlock.Header().Signatories()
case block.Base:
        if rebaser.scheduler != nil {
                rebaser.scheduler.Rebase(rebaser.expectedRebaseSigs)
        }
        rebaser.expectedKind = block.Standard
        rebaser.expectedRebaseSigs = nil
}
```

In case of a `rebase` or `base` block the `expectedKind` for the next block and the `expectedRebaseSigs` are set. This is important as otherwise the next block will be rejected.

In case of fast-forwarding, `syncLatestCommit()` only inserts the block but the observer is not notified and consequently these values are not updated. Thus, in case of fast-forwarding to a `rebase` block, the `base` block of the `propose` message will be rejected.

Code corrected:

The new code version of `syncLatestCommit()` notifies the observer in case the fast-forwarding succeeds:

```
if p.observer != nil {
        p.observer.DidCommitBlock(latestCommit.Block.Header().Height())
}
```

6.10 Incorrect precommits interfere with fast-forward functionality Design Medium

Code Corrected

The scheduled proposer creates a proposal for the new block. A proposal includes a struct entry called `latestCommit`, which enables nodes that have fallen out of sync to receive the latest block and sufficiently many precommits endorsing this block.

While crafting a new propose, this struct entry `latestCommit` is constructed and the precommits are added. The implementation of this, together with other issues is problematic:

```
messages := p.state.Precommits.QueryMessagesByHeightWithHighestRound(p.state.CurrentHeight - 1)
commits := make([]Precommit, 0, len(messages))
for _, message := range messages {
        commit := message.(*Precommit)
        if commit.blockHash.Equal(previousBlock.Hash()) {
                commits = append(commits, *commit)
        }
}
propose.latestCommit = LatestCommit{
        Block:      previousBlock,
        Precommits: commits,
}
```

The precommits are fetched using `QueryMessagesByHeightWithHighestRound()`. This function is defined in `messages.go` and fetches the precommits with the highest round for this block height present in the Inbox.

Due to the issue that all received messages (which are signed by a valid signatory) are stored in the Inbox, an adversarial signatory can craft and broadcast precommit messages for specific heights and specific rounds. These messages will be inserted into the precommit inbox.

This allows an adversarial signatory to insert precommit messages at a very high round. Consequently the function `QueryMessagesByHeightWithHighestRound()` used in the code above will return his precommit message only.

Due to the check `if commit.blockHash.Equal(previousBlock.Hash())` which will most certainly be false (assuming the adversarial signatory can not guess a future blockhash or insert the message on very short notice once the blockhash is known) the resulting `LatestCommit` struct will contain no precommit.

The consequence will be that nodes which are out of sync and see this proposal cannot fast-forward as expected. Hence, a single adversarial signatory can block fast-forwarding for all other nodes.

Side notes: First, the code implicitly expects that there are enough valid precommits. To decrease network overhead, only `2f+1` precommits could be included to endorse the block. Secondly, there is no check built in if the crafted struct `latestCommit` is actually valid before the message is broadcasted.

Code corrected: `QueryMessagesByHeightWithHighestRound()` has been changed and now only considers rounds for which the inbox contains `2f` messages. An adversarial signatory can no longer interfere with this function by inserting a precommit message at a very high round. Additionally the resulting `LatestCommit` will now only contain `2f+1` precommits.

6.11 Malformed input data causes panic Security

Design Medium Code Corrected

Input validation is generally missing when unmarshalling different datatypes (e.g. blocks, block headers or messages etc.). Simple fuzzing reveals various inputs which cause the unmarshalling to crash.

As this is a fairly general finding, we do not mention each occurrence separately. Instead, we provide examples code that is supsectible to crash:

`block/marshal.go` line 245:

```
if err := binary.Read(buf, binary.LittleEndian, &numBytes); err != nil {
        return fmt.Errorf("cannot read block.header len: %v", err)
}
headerBytes := make([]byte, numBytes)
```

`make()` crashes if `numBytes` is to large with and *out of memory* error, due to an overly large memory allocation. A similar issue exists for `process/marshal.go` on line 205. Other crashes occur due to *panic: runtime error: makeslice: len out of range*.

Code corrected: Marshalling and unmarshalling of data is now handled using the *surge* library. This library never explicitly panics and protects against malicious inputs.

6.12 No catch up after missing base block

Design Medium Specification Changed

In case a node was offline for some time it can use fast-forwarding to catch up to the current state. However, this does not work in case a base block occurred in between as it cannot verify the new signatories. The `Resync` message can be used to replay previous messages, but if the base block has been succeeded by two more blocks, it will no longer be included in those messages.

Hence, a node that misses as few as three blocks, which happen to include a base block, cannot really catch up any longer.

Specification changed:

The specification was clarified. In particular, it was clarified that message delivery has to be guaranteed for propose and precommits that are related to base blocks.

6.13 Precommits may be blocked if proposal arrives last Design Medium Code Corrected

In summary, there are three findings around this issue:

1. The text describing the function is incorrect
2. Emitting precommits may be blocked for the case when the proposal arrives last
3. `First` in height x round y markers may trigger actions on other block heights and rounds

In the following, we describe each of these findings in detail.

1. The `checkProposeInCurrentHeightAndRoundWithPrevotesForTheFirstTime` function is responsible to send the precommit endorsing a proposal, if the conditions to do so have been reached. It is annotated with:

   ```
   // checkProposeInCurrentHeightAndRoundWithPrevotesForTheFirstTime must only be
   // called when a Propose and 2f+1 Prevotes has been seen for the first time at
   // the current `block.Height` and `block.Round`. This can happen when a Propose
   // is seen for the first time at the current `block.Height` and `block.Round`,
   // or, when a Prevote is seen for the first time at the current `block.Height`
   // and `block.Round`.
   ```

   This states, this function must only be called when a `Propose` and `2f+1` prevotes has been seen for the first time at the current `block.Height` and `blockRound`. The current implementation does not respect this, the function itself however enforces these conditions. The description is incorrect and should be amended to reflect the actual implementation.

Under normal conditions, the proposal arrives before the prevotes and the handling of the prevotes should trigger the precommit message once sufficient prevotes for a blockhash have been received. However, as messages generally may arrive out-of-order, more than `2f` prevotes may arrive before the actual proposal. In this case the precommit message must be emitted after processing the proposal.

`checkProposeInCurrentHeightAndRoundWithPrevotesForTheFirstTime` gets called twice in the codebase:

- `handlePrevote()` calls this function whenever it has received more than `2f` prevotes on a blockhash at a certain height and round
- `handlePropose()` calls this function whenever it received a proposal for the `firstTime` at any height & round

Note that the `checkProposeInCurrentHeightAndRoundWithPrevotesForTheFirstTime` does not take any argument. Regardless of the height and round of the processed message, this function always operates on the current height and round of the node.

The function does the following:

- The proposal of the scheduled proposer is loaded. The function returns if the proposal is not available. This is in case more than `2f` prevotes arrived, but no proposal arrived.
- The amount of prevotes for this proposal's blockhash is fetched and checked.

- Only if there are more than `2f` prevotes for this blockhash and the state is in `StepPrevote` a precommit for this hash is emitted.

Hence there are two findings:

2. In case another proposal and `2f+1` prevotes arrived before the correct proposal, the correct proposal is not processed. Hence, no precommit is emitted, even though the valid conditions are fulfilled.
3. Messages with the wrong height or round can trigger `checkProposeInCurrentHeightAndRoundWithPrevotesForTheFirstTime` to be executed for the current round. The arrival of such messages of influences the execution of `checkProposeInCurrentHeightAndRoundWithPrevotesForTheFirstTime`.

Code corrected:

All three findings have been addressed by changing the code:

- The comments have been enhanced.
- The function `handlePropose()` has been enhanced, the updated implementation only inserts the proposal of the scheduled proposer into the Inbox. This ensures that the `first` marker triggers correctly which resolves the issue described above where precommits might be blocked.
- `First` markers triggered by messages of other heights than the nodes `current.height` no longer trigger actions on the nodes `current.height` as all calls to `checkProposeInCurrentHeightAndRoundWithPrevotes()`, `checkProposeInCurrentHeightAndRoundWithPrevotesForTheFirstTime()` and `checkProposeInCurrentHeightWithPrecommits()` have been wrapped into `if` branches which are only executed if the message being processed is at the same height.

Note that the new `HandleMessage()` implementation of `replica.go` which now features a message queue and drops outdated messages already ensures that messages are only processed in order and when the message height is equal to the `current.height`, therefore these checks are redundant. There is only one exception: The message queue stores and executes all received messages of future heights. In case `>2f+1` messages for a future height arrive before the node moves to this height, these messages have already been pushed to the queue and will be executed. After `2f+1 precommit` messages the node will move on to the next height/round but will still execute the remaining queued messages at the previous height and round, which are not dropped in this case. This however would be no problem as these messages will not trigger any action.

6.14 Shard not included in signature Security Medium Code Corrected

Messages inside `replica.go` are defined as:

```
type Message struct {
        Message process.Message
        Shard   Shard
}
```

The signature is stored inside `Message` and only calculated over the hash of the inner `Message`. The shard identifier is not included in the signature.

This can also be seen in the implementation of `broadcast.go` where these messages are sent:

```
func (broadcaster *signer) Broadcast(m process.Message) {
    if err := process.Sign(m, broadcaster.privKey); err != nil {
            panic(fmt.Errorf("invariant violation: error broadcasting message: %v", err))
    }
    broadcaster.broadcaster.Broadcast(Message{
            Message: m,
            Shard:   broadcaster.shard,
    })
}
```

Only the message is signed, not the shard identifier. Hence, any network-based attacker could simply change the shard information.

Code corrected: Additionally to the already signed inner `Message`, a field `Signature` has been added to the `Message` struct.

```
type Message struct {
        Message process.Message
        Shard   Shard
        Signature id.Signature
}
```

To calculate the signature, the message is marshalled into it's binary representation. All fields of the message are covered, this resolves the issue mentioned above.

6.15 Single-time trigger can be triggered twice

Security Medium Code Corrected

The `Insert` method is used for tracking the different conditions under which the state machine is allowed to transition between various states. It's correctness is fundamental to the correctness of the overall implementation. Among other return values, `Insert` returns the value `firstTimeExceeding2FOnBlockHash`, which signals that the critical threshold of 2f has been passed.

`firstTimeExceeding2FOnBlockHash` should only be true the first time that more than `2f` unique messages have been seen for the same blockhash (at a given height & round).

However, `true` might be returned for `firstTimeExceeding2FOnBlockHash` multiple times for the same height & round. Consider the following scenario:

1. 2f+1 messages for the same blockhash `0xA` arrive. Hence, when inserting the last message, `firstTimeExceeding2F0nBlockHash` will be true.

2. Later one of these 2f+1 signatories might change his blockhash. He does this by sending a new message for this height and round with another blockhash `0xB`.
3. So that `firstTimeExceeding2FOnBlockHash` return true for a second time at this height and round, one of the remaining signatories sends a message for this blockhash `0xA`.

   This last step occurs for the following reasons. The code checks for an existing message:

   ```
   _, ok := inbox.messages[height][round][signatory]
   ```

   where `ok` will be false because there was no entry yet for that signatory, hence the following branch is executed:

   ```
   if !ok {
       nOnBlockHash = inbox.QueryByHeightRoundBlockHash(height, round, message.BlockHash())
   }
   ```

   This counts the number of messages for the blockhash again, which will again be 2f+1.

   Finally, `firstTimeExceeding2FOnBlockHash` is returned as

   ```
   firstTimeExceeding2FOnBlockHash = !ok && (nOnBlockHash == 2*inbox.F()+1)
   ```

   which evaluates to true.

Code corrected: Signatories can no longer overwrite previously sent messages:

```
conflicting, ok := inbox.messages[height][round][signatory]
if ok {
        // We do not override existing messages. This means that it is
        // impossible for N to change, and thus for any "first time" triggers to
        // become true.
        if conflicting.SigHash().Equal(message.SigHash()) {
                // Messages are exact duplicates of each other.
                return previousN, false, false, false, false, nil
        }
        // Messages are in conflict.
        return previousN, false, false, false, false, conflicting
}

inbox.messages[height][round][signatory] = message
```

Only if no message is present in `inbox.messages[height][round][signatory]` the new message is stored in the inbox. This prevents the issue described above.

6.16 Checking the `BaseHash` as part of block validity Design Low Code Corrected

During the execution of the `isBlockValid` function inside `rebase.go` the `BaseHash` of the to-be-checked block is compared to the latest locally stored base block. However, currently this check only occurs if `checkHistory` is `true`.

Under the assumption that base blocks are not missed, it can always be checked.

Code corrected:

The code was corrected accordingly. The check is always performed in the new version.

6.17 Consistent handling for commits Design Low Code Corrected

When a new base block is committed through the regular procedure, a `Resync` message is sent out. However, if a base block is committed through the `syncLatestCommit` function no `Resync` messages is broadcasted.

Code corrected: The new code is consistent in handling the commits of base blocks.

6.18 Different Blocks can have same timestamps Security Low Code Corrected

The validity check for block timestamps, rejects a new block if it fulfills the following condition:

```
if proposedBlock.Header().Timestamp() < parentBlock.Header().Timestamp() {
```

Hence, equality of timestamps is allowed for different blocks. These might lead to complications on higher-level protocols, as most blockchain protocols, e.g. Ethereum, do not allow identical timestamps for different blocks.

Code corrected: The validity check is now has the following rejection criteria:

```
if proposedBlock.Header().Timestamp() <= parentBlock.Header().Timestamp() {
```

Hence, the issue is resolved.

6.19 Inconsistent handling in Message Queue

Correctness Low Code Corrected

The message queue stores messages that have arrived but cannot be processed yet. During `PopUntil` the `ProposeMessageType` is cleared, but the `ResyncMessageType` is not cleared. Both types should never be part of the queue.

Code corrected: The `ResyncMessageType` is now also cleared.

6.20 No round check performed Security Low

Specification Changed

When a new proposal is received, a number of tests are performed. These include checks whether the round of the proposal matches the expected round. However, a proposal contains two round entries. One directly inside the proposal. This one is checked as mentioned above. Another round entry is inside the `Block` that is included in the proposal.

This second entry is neither checked when parsing the proposal, nor during the validation of `IsBlockValid`. Hence, honest nodes could accept wrong round values.

Specification Changed:

Ren clarified that the `round` of the `Block` has a different meaning. The code contains the updated comment:

```
height       Height    // Height at which the block was proposed (and committed)
round        Round     // Round at which the block was proposed
```

6.21 One Signatory only Design Low Code Corrected

`replica.go` implements the `New()` method on the `Replica  struct`. To ensure that there are always `3f+1` signatories, there is following check:

```
if len(latestBase.Header().Signatories())%3 != 1 {
        panic(fmt.Errorf("invariant violation: number of nodes needs to be 3f +1, got %v", len(latestBase.Header().Signatories())))
}
```

Later `f` is initialized as:

```
process.DefaultState((len(latestBase.Header().Signatories())-1)/3),
```

The check does not prevent one signatory which would result in an `f` of zero.

Code corrected:

The updated code now enforces a minimum of four signatories in both places where the signatories are set:

- During the creation of a new replica
- When updating the signatories during a rebase

6.22 Possible Blockhash Collision Security Low Code Corrected

The blockhash is computed as follows:

```
func ComputeHash(header Header, txs Txs, plan Plan, prevState State) id.Hash {
    return sha256.Sum256([]byte(fmt.Sprintf("BlockHash(Header=%v,Txs=%v,Plan=%v,PreviousState=%v)", header, txs, plan, prevState)))
}
```

As Hyperdrive is generally agnostic to the structure of the underlying transactions, plans and states, their structure is not known. In particular, on the Hyperdrive-level, `plan` and `prevState` are essentially a byte array. Hence, the following collision could theoretically happen.

First Example Block:

```
plan = ",PreviousState="
prevstate = ""
```

Second Example Block:

```
plan = ""
prevstate = ",PreviousState="
```

Both blocks would have the same blockhash, as they would have the same string representation, even though they have different contents.

Code Corrected:

The block hash is now computed differently: The components of the block are now marshalled using the surge library into a bytes buffer, the hash is then calculated over this buffer.

The surge library contains the following documentation:

```
// When marshaling arrays/slices/maps, an uint32 length prefix is marshaled and
// prefixed.
```

Hence, the issue is avoided as length prefixes are added.

6.23 Signatory can DoS using fast-forward

Security Low Code Corrected

Any signatory can craft a malicious propose message to put a lot of computational load on other nodes.

By crafting a proposal for a block far in the future and including a large list of precommits, all signed by itself, a signatory can trigger a large number of cryptographic signature verifications, which are computationally expensive.

During the execution of the `syncLatestCommit()` function, the Hyperdrive loops over all precommits and verifies their signature before checking the uniqueness of the signatories after the loop. Only then is it detected that the propose message does not have sufficiently many unique signatories and the execution of `syncLatestCommit()` is stopped.

Code corrected: This is no longer possible, the `syncLatestCommit()` function has been changed to only accept `LatestCommit` structs containing exactly `2f+1 precommits`.

6.24 TODO in Code Design Low Code Corrected

There is a remaining `TODO` in `rebase.go` on line 139:

```
// TODO: Transactions are expected to be nil (the plan is not expected
// to be nil, because there are "default" computations that might need
// to be done every block).
```

Code corrected: The comment has been removed as this is not necessarily true and does not affect rebasing.

6.25 Unfair Round-Robin Scheduler Suggestion

Code Corrected

The current `RoundRobinScheduler` which determines the next proposer has some drawbacks.

Whenever a proposer fails to propose on his turn, the next signatory is tasked to proceed with a proposal. The signatories are selected with the following scheme:

```
scheduler.signatories[(uint64(height)+uint64(round))%uint64(len(scheduler.signatories))]
```

This has two consequences:

1. If a single signatory goes offline, it's always the same signatory "replacing" them. This basically doubles the proposal power of the replacing signatory, which is neither justified nor fair, as the documentation states that voting power should be equally distributed.
2. If a single signatory *x* misses its turn, then signatory *x+1* can propose two subsequent blocks. The one where it replaces the original signatory and the next one, which is its scheduled block to propose.

Code corrected: The unfairness of the round-robin is only really relevant when either block rewards are given to the proposer or there is no punishment for failing to propose repeatedly. To support general use cases the scheduler can now be replaced.

1	https://arxiv.org/pdf/1807.04938.pdf
2(1, 2)	https://arxiv.org/pdf/2004.10617.pdf