Uniswap Rebate
Security Review

Cantina Managed review by:

R0bert, Lead Security Researcher
Haxatron, Security Researcher

June 13, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Uniswap is an open source decentralized exchange that facilitates automated transactions between ERC20 token tokens on various EVM-based chains through the use of liquidity pools and automatic market makers (AMM).

From May 21st to May 24th the Cantina team conducted a review of uniswap-rebate on commit hash 903c5424. The team identified a total of **19** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	2	2	0
Medium Risk	9	7	2
Low Risk	6	4	2
Gas Optimizations	0	0	0
Informational	2	1	1
Total	**19**	**14**	**5**

3 Findings

3.1 High Risk

3.1.1 Prover can panic if a proving request exceeds `MaxSwapNum` or `MaxPoolNum` limit

Severity: High Risk

Context: provereq.go#L48-L60, prove.go#L126-L141

Description: It is possible for the proving request to exceed the `MaxSwapNum` or `MaxPoolNum` limit. This can occur when a single block contains more than the `MaxSwapNum` or `MaxPoolNum` limit.

If we take a look at the batching logic to observe what happens, the `else` branch is executed and the block containing more than the `MaxSwapNum` or `MaxPoolNum` is added to the `curReq`. This block is never really discarded and the next block processed will result in the `else` branch to be executed again causing the `curReq` to be saved into `proveReqs`, thus creating a prove request that contains more than the `MaxSwapNum` or `MaxPoolNum` limit.

```
for _, blknum := range blkNums {
    // pupulate curReq blk map
    block, _ := c.ec.BlockByNumber(context.Background(), new(big.Int).SetUint64(blknum))
    curReq.Blks[blknum] = OneBlock{
        BaseFee:   block.BaseFee().Uint64(),
        Timestamp: block.Time(),
    }
    one := blk2swaps[blknum]
    // swaps includes logs and map of poolid, if within limit, add to curReq
    if len(curReq.Logs)+len(one.Logs) <= circuit.MaxSwapNum &&
        curPoolMap.CombineCount(one.PoolIds) <= circuit.MaxPoolNum {
        curReq.Logs = append(curReq.Logs, one.Logs...)
        curPoolMap.Merge(one.PoolIds)
    } else {
        // can't fit, need to create new req, but first populate req.PoolKey
        for k := range curPoolMap {
            curReq.PoolKey = append(curReq.PoolKey, poolidMap[k])
        }
        proveReqs = append(proveReqs, curReq)
        curReq = c.NewOneProveReq(&claimev.Raw)
        curPoolMap = make(PoolIdMap)
        curReq.Logs = append(curReq.Logs, one.Logs...)
        curPoolMap.Merge(one.PoolIds)
    }
}
```

Following this, the prove request is submitted to the prover, and during circuit construction all the swaps (`r.Logs`) and pools contained in the prove request `r.PoolKey` is iterated through.

```
for i, poolkey := range r.PoolKey {
    idx := i * 5
    // todo: break if idx > len(ret.PoolKey)
    ret.PoolKey[idx] = sdk.ConstFromBigEndianBytes(poolkey.Currency0[:])
    ret.PoolKey[idx+1] = sdk.ConstFromBigEndianBytes(poolkey.Currency1[:])
    ret.PoolKey[idx+2] = sdk.ConstFromBigEndianBytes(poolkey.Fee.Bytes())
    ret.PoolKey[idx+3] = sdk.ConstFromBigEndianBytes(poolkey.TickSpacing.Bytes())
    ret.PoolKey[idx+4] = sdk.ConstFromBigEndianBytes(poolkey.Hooks[:])
}
// skip first Claim ev
for i, swap := range r.Logs[1:] {
    ret.TxGasCap[i] = sdk.ConstUint32(swap.TxGasCap)
}
```

Since `ret.PoolKey` and `ret.TxGasCap` (the circuit inputs) are fixed size arrays, iterating through more than `MaxSwapNum` or `MaxPoolNum` limit will cause array out-of-bounds error to occur and the program to panic. Furthermore, `r.Logs` (that can contain more than `MaxReceipts`) will be iterated through as well. Making it entirely possible that `AddReceipt` is called more than `MaxReceipts` number of times, making the number of receipts go over circuit limits.

```
for i, onelog := range r.Logs {
    blkNum := onelog.Swap.BlockNumber
    block := r.Blks[blkNum]
    if blkNum != lastBlockNum {
        app.AddStorage(sdk.StorageData{
            BlockNum:     new(big.Int).SetUint64(blkNum),
            BlockBaseFee: new(big.Int).SetUint64(block.BaseFee),
            Address:      onchain.Hex2addr(r.Oracle),
            Slot:         onchain.ZeroHash,
            Value:        block.SlotValue,
        }, i) // special mode, sdk will fill dummy in between
    }
    rd := OneLog2SdkReceipt(onelog, block.BaseFee)
    receipts = append(receipts, &rd)
    app.AddReceipt(rd)
}
```

Recommendation: Consider either preventing blocks that exceed the limits from being added to prove requests and thus proven (with the trade-off that rebates for swaps in these blocks will not be able to claimed).

Or handle gracefully in the circuit construction if `idx > ret.PoolKey` or `i > ret.TxGasCap` as well as `r.Logs > MaxReceipts` when adding the receipts and thereby ignoring swaps in the block that would exceed the `MaxPoolNum` and the `MaxSwapNum` limit.

Uniswap Labs: Fixed in commit 0eaffe63.

Cantina Managed: Fix verified.

3.1.2 Risk of `reqid` collisions leading to misrouted proof retrieval

Severity: High Risk

Context: server.go#L41

Description: The ZK application generates a request ID (`reqid`) simply based on the current epoch time in seconds. If multiple requests arrive during the same second, different users could end up sharing the same `reqid`. Once created, that ID is used in later steps, including the `GET /zk/get/{reqid}` endpoint, to retrieve proof data. This potential overlap causes an ambiguity: the second user's query might unintentionally pick up the proof for the first user, or vice versa, returning a proof that belongs to another set of transactions and, therefore, to another `router` and `claimer`. That mismatch can lead to confusion and failed calls to the `RouterRebates.claimWithZkProof` function.

Recommendation: Generate a universally unique request identifier instead of a raw epoch second. For instance, combine a high-resolution timestamp with a random or monotonically increasing counter, or use a UUID, so each request produces a guaranteed distinct ID. This ensures that the `GET /zk/get/{reqid}` endpoint consistently returns the correct proof for only its rightful owner.

Uniswap Labs: There is no risk of `reqid` collision in the database as schema.sql#L23 `reqid` is used as a primary key that must be unique. Accepting and generating zk proof takes time so we always have a Nginx config with rate limit. Fixed in commit 6399484f.

Cantina Managed: Fix verified.

3.2 Medium Risk

3.2.1 No limit on base fee allows abuse of the rebate program to perform block stuffing attacks

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: Block stuffing attacks primarily work by stuffing blocks filled with useless transactions to prevent other users from using the chain. This is prevented by EIP-1559, where the block base fee increases exponentially by 12.5% from the previous block based on the much gas was used in the previous block.

The Uniswap rebate program refunds gas per swap based on the formula (where `n` being the number of swaps in the transaction):

```
totalRebate = min(80000 * n + 80000, 0.8 * gasUsed) * blockBaseFee
```

The `blockBaseFee` is the block where the the swap was executed in. Since there is no limit on the `blockBaseFee` enforced in the rebate program, a malicious attacker can abuse the rebate program to subsidise block stuffing attacks. They can do this by sending multiple transactions performing "*hyperoptimized swaps*" ensuring that the total gas refunded is 80% of the cost of the transaction. Essentially, the rebate program can be abused to conduct block stuffing attacks on chains at 80% cheaper, and at the same time drain the rebate program of its funds.

Recommendation: Enforce a cap on the `blockBaseFee` used to prevent abuse of the rebate program.

Uniswap Labs: Fixed in commit 089d87d4.

Cantina Managed: Fix verified.

3.2.2 Last prove request is accidentally discarded

Severity: Medium Risk

Context: onchain.go#L164-L200

Description: Swap requests in each block are batched into prove requests that have a limit of `MaxSwapNum` and `MaxPoolNum`. In the code referenced in `onchain.go`, only when the limit is exceeded in the `else` branch if a block of swap requests are added will the `curReq` be added to the `proveReqs` array.

```
} else {
    // can't fit, need to create new req, but first populate req.PoolKey
    for k := range curPoolMap {
        curReq.PoolKey = append(curReq.PoolKey, poolidMap[k])
    }
    proveReqs = append(proveReqs, curReq)
    curReq = c.NewOneProveReq(&claimev.Raw)
    curPoolMap = make(PoolIdMap)
    curReq.Logs = append(curReq.Logs, one.Logs...)
    curPoolMap.Merge(one.PoolIds)
}
```


The problem is that for the last batch of swap requests that do not exceed the `MaxSwapNum` and `MaxPoolNum` limit, only the `if` branch will be executed but the `curReq` will not be added to the `proveReqs` array and consequently will not be forwarded to the prover. As a result, due to the missing update to the `proveReqs` array when the limit is not exceeded, any swaps contained in the last batch are accidentally discarded and the user has to submit these transactions to the ZK application again.

Recommendation: Add the `curReq` to `proveReqs` after the last block is processed.

```
  for _, blknum := range blkNums {
      // pupulate curReq blk map
      block, _ := c.ec.BlockByNumber(context.Background(), new(big.Int).SetUint64(blknum))
      curReq.Blks[blknum] = OneBlock{
          BaseFee:   block.BaseFee().Uint64(),
          Timestamp: block.Time(),
      }
      one := blk2swaps[blknum]
      // swaps includes logs and map of poolid, if within limit, add to curReq
      if len(curReq.Logs)+len(one.Logs) <= circuit.MaxSwapNum &&
          curPoolMap.CombineCount(one.PoolIds) <= circuit.MaxPoolNum {
          curReq.Logs = append(curReq.Logs, one.Logs...)
          curPoolMap.Merge(one.PoolIds)
      } else {
          // can't fit, need to create new req, but first populate req.PoolKey
          for k := range curPoolMap {
              curReq.PoolKey = append(curReq.PoolKey, poolidMap[k])
          }
          proveReqs = append(proveReqs, curReq)
          curReq = c.NewOneProveReq(&claimev.Raw)
          curPoolMap = make(PoolIdMap)
          curReq.Logs = append(curReq.Logs, one.Logs...)
          curPoolMap.Merge(one.PoolIds)
      }
+     for k := range curPoolMap {
+         curReq.PoolKey = append(curReq.PoolKey, poolidMap[k])
+     }
+         proveReqs = append(proveReqs, curReq)
  }
```

Uniswap Labs: Fixed in commits 0eaffe63 and ae961a32.

Cantina Managed: Fix verified.

3.2.3 Reorgs are not handled

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: As noted in a separate audit for the web application for Uniswap rebates, there is no check that a transaction containing swaps has been reorg-ed out of the chain. This means a user can obtain refunds for swaps that has reorg-ed out the chain.

The corresponding fix for the web application is here in PR 45 enforcing a minimum distance between the latest block number and the block number of the transactions submitted to the application. There is no such check for the ZK application.

Recommendation: Apply the same fix from the web application to the ZK application, enforcing a minimum distance between the latest block number and the block number of the transactions submitted.

Uniswap Labs: Fixed in commit a6a1fee9.

Cantina Managed: Fix verified.

3.2.4 Inability to update the claimer address once inserted into the database

Severity: Medium Risk

Context: queries.sql#L23-L24

Description: The current database logic for storing `Claimer` events does not allow for updating an existing `claimer` address once it has been set. Specifically, the `INSERT INTO claimer (chid, router, evlog) VALUES ($1, $2, $3)` statement lacks an `ON CONFLICT` or `UPSERT` clause. As a result, when the same (chid, router) pair is inserted a second time, the database will throw a duplicate key error:

```
CREATE TABLE IF NOT EXISTS claimer (
    chid BIGINT NOT NULL, -- chainid
    router TEXT NOT NULL,  -- 0x... router address
    evlog JSONB NOT NULL, -- ClaimHelpClaimer struct (not types.Log b/c we need to add Value/Scan)
    UNIQUE (chid, router) -- ensure no duplicated router on same chain, auto-create index
);
```

This situation can occur if the `Router` emits a new `Claimer` event indicating an updated `claimer` address, or if an operator needs to correct prior data. Because the code never attempts an explicit update, the first record becomes permanent in the table. Consequently, any subsequent `Claimer` event for the same (chid, router) cannot be persisted, leaving the `Router` with an stale `Claimer` address.

Recommendation: Enhance the insertion logic with an `ON CONFLICT` clause (or similar upsert mechanism) to accommodate updates to an existing (chid, router) entry in the `claimer` table. By doing so, the row can be replaced or merged when a new `Claimer` event for the same (chid, router) is encountered. This ensures the database accurately reflects the current claimer address over time.

Uniswap Labs: Acknowledged. This is currently by design. A manual operation to update the internal database is required if a router sets a new claimer.

Cantina Managed: Acknowledged.

3.2.5 Discrepancy between ZK app's hardcoded rebate parameters and `RouterRebates` contract

Severity: Medium Risk

Context: config.toml#L1-L16

Description: The ZK application calculates rebated gas using parameters defined in the `server/config.toml` file (e.g., `GasPerSwap = 80000`). Meanwhile, the on-chain `RouterRebates` contract stores its own values (`rebatePerSwap`, `rebatePerHook`, `rebateFixed`) which can be changed at any time via setter methods. Because these two codebases rely on independent references for per-swap and per-transaction gas, they can easily diverge. If the `RouterRebates` contract is updated to reflect new values while the ZK app remains tied to a prior default in `config.toml`, the circuit will compute an out-of-date rebate amount. This results in a mismatch between the values used in `claimWithSignature` and `claimWithZkProof`, generating confusion and inconsistent user experiences for the same router.

Recommendation: Have the ZK application dynamically fetch the `rebatePerSwap`, `rebatePerHook` and `rebateFixed` values from the `RouterRebates` contract at runtime, instead of embedding them in static configuration. This ensures the logic in the circuit remains synchronized with the on-chain contract, eliminating discrepancies. If direct chain reads are costly or impractical, consider at least providing a secure update mechanism that regularly refreshes the ZK app's parameters whenever the on-chain contract's values change.

Uniswap Labs: Acknowledged. These parameters will not be frequently updated. When they are updated we will coordinate the config update with the Uniswap team. We will stop the server, wait for the on-chain config to update, update the config in our server and then start the server again.

Cantina Managed: Acknowledged.

3.2.6 Unbounded concurrent proof generation may lead to server instability

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The `prove.go` file accepts proof generation requests at the `POST /prove` endpoint and immediately spawns a new goroutine to handle each request (`go ProveReqs(req)`). Because no concurrency limit or rate-limiting mechanism is applied, a surge of incoming requests can trigger multiple simultaneous, CPU and memory-intensive proof computations. Proof generation is resource-heavy, causing exhaustion of system resources (e.g., CPU, RAM) and leading to denial-of-service if too many proofs are generated at once. An attacker, or even legitimate users in large numbers, could easily overwhelm the server simply by sending numerous concurrent requests, causing the server process to crash or become unresponsive.

Recommendation: Institute a concurrency management strategy to control how many proofs can be generated in parallel. Common approaches include:

- A worker pool or semaphore-based limit that queues requests once a certain threshold of ongoing proofs is reached.
- Server-side rate limiting on the `GET /zk/new` endpoint, ensuring a fair, bounded number of incoming requests.

Uniswap Labs: Fixed in commit 6399484f. Prover pools will be deployed for the production environment.

Cantina Managed: Fix verified.

3.2.7 Potential out-of-bounds panic if a transaction with empty logs is provided

Severity: Medium Risk

Context: onchain.go#L132-L138

Description: In `onchain.go`, for every transaction, before all the swaps are iterated the log index offset is first fetched from `r.Logs[0].Index`.

```
// go over tx receipts to filter eligible Swap logs
var logs []OneLog
for _, r := range receipts {
    // all logs in one receipt has same logIdxOffset
    logIdxOffset := r.Logs[0].Index
    hasAppend := false // for this receipt, whether we have appended OneLog to logs, if true we need to set
    ↪  last OneLog TxGasCap
    for _, l := range r.Logs {
```

However, if a transaction has no logs (meaning that no events were emitted), then doing this will result in an out-of-bounds array access which could potentially lead to a panic and crash if the panic is not recovered from (or potentially a `Internal Server Error` returned by the web server if the panic is indeed recovered), as there are no checks that `r.Logs` before accessing the first element via `r.Logs[0].Index`.

Recommendation: Check that `len(r.Logs)` is not 0 before accessing `r.Logs[0].Index`.

Uniswap Labs: Fixed in commit 0eaffe63.

Cantina Managed: Fix verified.

3.2.8 Potentially unclaimable swaps due to `FindSender` implementation

Severity: Medium Risk

Context: onchain.go#L203-L217

Description: Within `onchain.go`, the function `FindSender(receipts []*types.Receipt, ...)` simply returns the first recognized router address it finds in any of the transaction's swap logs:

```
func (c *OneChain) FindSender(receipts []*types.Receipt, poolids map[common.Hash]binding.PoolKey)
↪  common.Address {
    pmAddr := Hex2addr(c.PoolMgr)
    swapEvId := Hex2hash(SwapEvId)
    for _, r := range receipts {
        for _, l := range r.Logs {
            if l.Address == pmAddr && l.Topics[0] == swapEvId {
                if _, ok := poolids[l.Topics[1]]; ok {
                    return common.BytesToAddress(l.Topics[2][12:])
                }
            }
        }
    }
    return ZeroAddr
}
```

Any other routers that have subsequent swaps in the same transaction remain unrecognized. Because the code stops at the very first eligible router it finds, any subsequent swaps within the same transaction,

especially in an EIP-4337 user operation that may bundle multiple routers, will never be recognized as belonging to a second router. Consequently, the second router's swaps from that same transaction cannot be proved for a claim.

For example, consider a single transaction in block 1337 that executes:

- `swap1` for `Router1`.
- `swap2` for `Router2`.

But `FindSender` returns `Router1` upon encountering the first recognized swap event. Meanwhile, imagine `Router2` has already set `lastBlockClaimed[chainid][Router2] = 1337` in the `RouterRebates` storage (e.g. from a prior claim). When the ZK logic eventually tries to handle that second swap, it never sees `Router2` as the recognized "sender." Thus, even though Router2's swap occurred, it remains unaccounted for and effectively unclaimable. As `lastBlockClaimed[chainid][Router2]` is set to the block 1337 the workaround of taking a transaction from a previous block where Router2 is the first sender can not be executed.

Recommendation: Consider adding an extra parameter to the `GET /zk/new` endpoint that identifies the router address upon we want to execute the claim.

Uniswap Labs: Fixed in commit 4cd9b5ab.

Cantina Managed: Fix verified.

3.2.9 Potential service disruption due to Infura key quota exhaustion

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The ZK application relies on an Infura-provided RPC gateway to retrieve transaction receipts and block data for constructing proofs. This key is defined in the `config.toml` file. Under heavy load or concurrent requests, the allocated request quota may deplete, causing Infura to throttle or reject further calls. When this happens, the aggregator can't fetch real on-chain data, causing all the user GET requests to fail. Legitimate users then see system outages, incomplete proofs, or stuck requests until the key's usage resets or a larger plan is purchased. Because the ZK aggregator is fully dependent on a working RPC endpoint, any downtime or key exhaustion leaves the entire system unusable.

Recommendation: Adopt a robust strategy to handle RPC usage limits, such as:

- Monitoring request volumes and implementing rate limiting or queuing.
- Providing fallback endpoints or load-balanced keys across multiple providers (Infura, Alchemy, etc...).
- Automating key rotation or employing higher-tier plans to handle surges in usage.
- Operating an archive node to avoid third-party rate limits, giving the app complete independence from external providers.

Uniswap Labs: Fixed in commit 6399484f.

Cantina Managed: Fix verified.

3.3 Low Risk

3.3.1 `lastBlockNum` is not updated resulting in duplicate adds to storage

Severity: Low Risk

Context: prove.go#L125-L141

Description: In `prove.go` the `lastBlockNum` is missing an update and will always remain 0. This means that `blkNum != lastBlockNum` is always true and every swap in the same block will result in adding the same block information into storage over and over again.

```
var lastBlockNum uint64
for i, onelog := range r.Logs {
    blkNum := onelog.Swap.BlockNumber
    block := r.Blks[blkNum]
    if blkNum != lastBlockNum {
        app.AddStorage(sdk.StorageData{
            BlockNum:     new(big.Int).SetUint64(blkNum),
            BlockBaseFee: new(big.Int).SetUint64(block.BaseFee),
            Address:      onchain.Hex2addr(r.Oracle),
            Slot:         onchain.ZeroHash,
            Value:        block.SlotValue,
        }, i) // special mode, sdk will fill dummy in between
    }
    rd := OneLog2SdkReceipt(onelog, block.BaseFee)
    receipts = append(receipts, &rd)
    app.AddReceipt(rd)
}
```

Recommendation: Update the `lastBlockNum` as shown:

```
  var lastBlockNum uint64
  for i, onelog := range r.Logs {
      blkNum := onelog.Swap.BlockNumber
      block := r.Blks[blkNum]
      if blkNum != lastBlockNum {
          app.AddStorage(sdk.StorageData{
              BlockNum:     new(big.Int).SetUint64(blkNum),
              BlockBaseFee: new(big.Int).SetUint64(block.BaseFee),
              Address:      onchain.Hex2addr(r.Oracle),
              Slot:         onchain.ZeroHash,
              Value:        block.SlotValue,
          }, i) // special mode, sdk will fill dummy in between
+         lastBlocknum = blkNum
      }

      rd := OneLog2SdkReceipt(onelog, block.BaseFee)
      receipts = append(receipts, &rd)
      app.AddReceipt(rd)
  }
```

Uniswap Labs: Brevis SDK now populates block info automatically for receipt, see main/sdk/app.go#L1252-L1268. Related code handling block information was removed in commit 0eaffe63.

Cantina Managed: Fix verified.

3.3.2 Block information of blocks that exceed the limit are not added to the prove request

Severity: Low Risk

Context: onchain.go#L164-L201

Description: Swap requests in each block are batched into prove requests that have a limit of `MaxSwapNum` and `MaxPoolNum`. In the code referenced in `onchain.go`, if a block containing swaps were to exceed the limit of `MaxSwapNum` and `MaxPoolNum`, the `curReq` is added to `proveReqs` and is then reset by `curReq = c.NewOneProveReq(&claimev.Raw)`.

```
} else {
    // can't fit, need to create new req, but first populate req.PoolKey
    for k := range curPoolMap {
        curReq.PoolKey = append(curReq.PoolKey, poolidMap[k])
    }
    proveReqs = append(proveReqs, curReq)
    curReq = c.NewOneProveReq(&claimev.Raw)
    curPoolMap = make(PoolIdMap)
    curReq.Logs = append(curReq.Logs, one.Logs...)
    curPoolMap.Merge(one.PoolIds)
}
```

The problem is that the information of the block that exceeded the limit and included in the next batch is not included in the new `curReq.Blks[blknum]` map due to the missing update. This information is required

for the prover to determine the block base fee. Consequently, the block information is also included in the previous batch even when it is not supposed to be (as the block was included in the next batch and not the previous one due to exceeding limit).

Recommendation: Instead of populating the block information in the `curReq` at the beginning everytime a new block is processed, the block information should be populated ONLY when the block is appended as shown. This solves both issues.

```
  var proveReqs []*OneProveReq
  curReq := c.NewOneProveReq(&claimev.Raw)
  curPoolMap := make(PoolIdMap) // unique poolids in current req
  // blkNums is sorted ascending
  blkNums, blk2swaps := SwapsByBlock(logs)
  for _, blknum := range blkNums {
-     // pupulate curReq blk map
-     block, _ := c.ec.BlockByNumber(context.Background(), new(big.Int).SetUint64(blknum))
-
-     // create new reqs, get the base fee and block time
-     curReq.Blks[blknum] = OneBlock{
-         BaseFee:   block.BaseFee().Uint64(),
-         Timestamp: block.Time(),
-     }
      // obtain the SSB corresponding to the block number

      one := blk2swaps[blknum]

      // swaps includes logs and map of poolid, if within limit, add to curReq

      // If <= maxSwapNum
      // AND <= MaxPoolNum
      if len(curReq.Logs)+len(one.Logs) <= circuit.MaxSwapNum &&
          curPoolMap.CombineCount(one.PoolIds) <= circuit.MaxPoolNum {

          curReq.Logs = append(curReq.Logs, one.Logs...)
          curPoolMap.Merge(one.PoolIds)
+         // pupulate curReq blk map
+         block, _ := c.ec.BlockByNumber(context.Background(), new(big.Int).SetUint64(blknum))
+         // create new reqs, get the base fee and block time
+         curReq.Blks[blknum] = OneBlock{
+             BaseFee:   block.BaseFee().Uint64(),
+             Timestamp: block.Time(),
+         }
      } else {
          // can't fit, need to create new req, but first populate req.PoolKey
          for k := range curPoolMap {
              curReq.PoolKey = append(curReq.PoolKey, poolidMap[k])
          }
          proveReqs = append(proveReqs, curReq)

          // Reset curReq and curPoolMap and add the SSB
          curReq = c.NewOneProveReq(&claimev.Raw)
          curPoolMap = make(PoolIdMap)

          curReq.Logs = append(curReq.Logs, one.Logs...)
          curPoolMap.Merge(one.PoolIds)
+         // pupulate curReq blk map
+         block, _ := c.ec.BlockByNumber(context.Background(), new(big.Int).SetUint64(blknum))
+         // create new reqs, get the base fee and block time
+         curReq.Blks[blknum] = OneBlock{
+             BaseFee:   block.BaseFee().Uint64(),
+             Timestamp: block.Time(),
+         }
      }
  }
  return proveReqs, nil
```

Uniswap Labs: Brevis SDK now populates block info automatically for receipt, see main/sdk/app.go#L1252-L1268. Related code handling block information was removed in commit 0eaffe63.

Cantina Managed: Fix verified.

3.3.3 Loading all eligible pools into memory could become unscalable for large datasets

Severity: Low Risk

Context: onchain.go#L113-L118

Description: When processing transactions in `ProcessReceipts()`, the code retrieves every pool record in the pools table and stores them in an in-memory map:

```
rows, _ := c.db.Pools(context.Background(), c.ChainID)
poolidMap := make(map[common.Hash]binding.PoolKey)
for _, row := range rows {
    poolidMap[Hex2hash(row.Poolid)] = row.Poolkey
}
```

With an estimation of around 10000 active pools with hooks, this approach can become a bottleneck. Each request loads the entire table into memory, leading to an overhead in CPU and RAM usage. Under concurrent load (multiple requests arriving at once), the server repeatedly rebuilds this large map, which can cause memory spikes. This situation worsens as more "Initialize" events accumulate in the database, continuously increasing the pool count.

Recommendation: Adopt a more selective strategy for verifying pool membership to prevent scaling issues. For example, rather than always building a full map, query the database for each encountered `poolId` to confirm eligibility. This way, even if thousands of pools exist and multiple concurrent calls arrive, resource usage stays manageable and the ZK app remains stable.

Uniswap Labs: Acknowledged. We tested the application with 100,000 generated pool records and no scalability issue was found.

Cantina Managed: Acknowledged.

3.3.4 `BlkInterval` / `BlkDelay` settings should align with the required blocks to avoid reorgs

Severity: Low Risk

Context: config.toml#L11-L12

Description: Within the ZK application's `config.toml` file, two parameters govern how block monitoring is performed:

- `BlkInterval`: The frequency (in seconds) at which the app checks for new blocks or logs. For example, `BlkInterval = 12` means the system polls every 12 seconds.
- `BlkDelay`: How many blocks behind "latest" the system considers safe to finalize. If `BlkDelay = 2`, it processes up to `currentBlock - 2` to avoid minor reorgs.

For example, on Ethereum mainnet, 12 seconds roughly equals one block on average and a delay of 2 blocks is insufficient to prevent reorgs. Moreover, `BlkInterval` and `BlkDelay` should align with the chain's typical block time. For instance, if `BlkDelay` is 15 blocks, you might want to set `BlkInterval` to (`15 * 12 = 180` seconds) to exactly poll every 15 blocks foe efficiency.

Recommendation:

- Pick a `BlkDelay` consistent with the level of reorg risk you want to mitigate. For Ethereum mainnet, 12 seconds is typical per block, but a delay of only 2 blocks does not really offer any reorg protection. The typical safe value for Ethereum mainnet would be 15 blocks.
- For each chain with different block times or reorg frequencies, tune `BlkInterval` and `BlkDelay` accordingly.

Uniswap Labs: Acknowledged. Config will be adjusted to match each supported chain.

Cantina Managed: Acknowledged.

3.3.5 Mismatch between circuit capacity and `MaxNumDataPoints`

Severity: Low Risk

Context: prove.go#L265

Description: The circuit is configured to handle up to 512 receipts (`MaxReceipts=512`), but in the gateway setup, `MaxNumDataPoints` is set to 128. This mismatch means the system is only declaring space for 128 data inputs to the gateway, while the circuit can handle many more. In scenarios where more than 128 receipts are actually toggled on, the gateway may not handle all of them correctly, leading to incomplete proofs or potential out-of-bounds references.

Although this may not trivially let an attacker forge data, it can cause serious problems:

- The circuit might interpret toggles for receipts beyond the 128th index, while the gateway only processes the first 128.
- A user could think they proved all 512 receipts, but only the first 128 are recognized by the external system, creating partial or incorrect verification results.

Recommendation: Ensure `MaxNumDataPoints` passed to the gateway and the local circuit capacity (`MaxReceipts + maxStorage + maxTransactions`) match exactly. On the other hand, before constructing proofs, confirm that the number of toggled data inputs does not exceed the declared maximum in your gateway request.

Uniswap Labs: Fixed in commit 72f7044b. Brevis gateway will honor `MaxReceipts`, but it's still a good idea to have numbers align properly.

Cantina Managed: Fix verified.

3.3.6 Incorrect usage of `MaxStorage` field

Severity: Low Risk

Context: prove.go#L263

Description: In the code snippet:

```
return &commonproto.AppCircuitInfoWithProof{
    // ...
    MaxReceipts: circuit.MaxReceipts,
    MaxStorage:  circuit.MaxReceipts, // <-- This should be MaxStorage, not MaxReceipts
    MaxTx:       0,
    MaxNumDataPoints: 128,
}
```

the `MaxStorage` field is incorrectly set to `circuit.MaxReceipts` instead of the intended `circuit.MaxStorage`. If other parts of the system rely on `MaxStorage` for controlling how many storage queries the circuit handles, then this mismatch can cause confusion, parsing errors, or silent truncation of storage data.

Recommendation: Use `circuit.MaxStorage` in the `MaxStorage` field so it accurately reflects the circuit's true capacity.

Uniswap Labs: Fixed in commit 2f5f2d96.

Cantina Managed: Fix verified.

3.4 Informational

3.4.1 `GasPerTx` is never initialized in the ZK app configuration

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In the ZK App's `server/config.toml` file, `GasPerTx` (the equivalent of a fixed per-transaction rebate) is never set. Therefore, it defaults to zero in the circuit's calculations which differs from the on-chain `RouterRebates` default config:

```
uint256 public rebateFixed = 80_000;
```

Consequently, the total rebate might omit any fixed per-tx component, yielding a mismatch between what the on-chain contract expects and what the ZK app computes. This will also cause that the amounts rebated between `claimWithSignature` and `claimWithZkProof` differ for the same set of transactions.

Recommendation: Add a `GasPerTx` entry to the config file so the ZK app accurately reflects the intended "rebateFixed" portion of gas rebates. Ensure this value matches the `rebateFixed` value set at the `RouterRebates` contract. This ensures consistent rebate logic between the on-chain contract and the off-chain circuit.

Uniswap Labs: Fixed in commit c24be0e1.

Cantina Managed: Fix verified.

3.4.2 Unbounded database growth from monitoring `Claimer` and `Initialize` events

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The ZK application continuously logs two types of events into the database:

- `Claimer` events from the `ClaimHelp` contract.
- Uniswap v4 `PoolManager` Initialization events.

Over time, as more and more new pools are deployed and additional Claimer events come in, these tables will grow indefinitely. Without any housekeeping or size management, the database can balloon in size and risk performance degradation, higher storage costs, or even complete exhaustion of storage resources. In the long run, this can disrupt the entire monitoring system, making queries and proof generation increasingly slow or leading to a database crash if capacity runs out.

Recommendation:

- Periodically archive or remove old records that are no longer relevant (e.g., Claimer events older than a certain timeframe, or inactive Uniswap v4 pools).
- If the application only needs to process the most actively traded or popular tokens, track a smaller subset of pools instead of every possible pool.
- Regularly track table sizes, generate alerts if they exceed thresholds and schedule maintenance tasks to clean or archive data.
- Consider using partitioned tables in the database, so older partitions can be offloaded more easily while keeping recent data fast to query.

Uniswap Labs: Acknowledged. Considered low priority as the DB can handle million of records.

Cantina Managed: Acknowledged.