PUBLIC

Code Assessment

of the CairoLib

November 10, 2023

Produced for

by

CHAINSECURITY

Contents

1 Executive Summary

Dear all,

Thank you for trusting us to help Herodotus with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of CairoLib according to Scope to support you in forming an opinion on their security risks.

CairoLib is a library primarily for Herodotus on Starknet implementing Ethereum related operations including Recursive Length Prefix (RLP) decoding, Keccak256 and Poseidon hash function wrappers, Merkle Patricia Trie (MPT) verification and Merkle Mountain Range (MMR) structures.

The most critical subjects covered in our audit are functional correctness, data integrity and consistency, and security vulnerabilities. Amongst others, the following issues have been uncovered:

- Missing Length Validation in MPT Verify
- MMR: Incorrect Root Update Possible, Insufficient Peaks Validation
- Keccak Discards Leading Zero Bytes in Last Little Endian Words64

After the intermediate report all issues have been resolved.

The general subjects covered are usability, efficiency and robustness.

In summary, for its intended usage in herodotus-on-starknet we find that the codebase of CairoLib provides a good level of security. However, it's worth noting that more thorough testing could have identified most of these issues early. Moreover, there is still room for enhancement in the testing processes.

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

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

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

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

Critical-Severity Findings	1
• Code Corrected	1
High-Severity Findings	2
• Code Corrected	2
Medium-Severity Findings	6
• Code Corrected	6
Low-Severity Findings	2
• Code Corrected	2

2 Assessment Overview

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

2.1 Scope

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

V	Date	Commit Hash	Note
1	19 September 2023	99b1c0e3b72e78a9db97e9c9e1a644304efc0e1f	Initial Version
2	30 October 2023	0af61150a5f738dbc1b3cdf0b477ed9643e94472	After Intermediate Report
3	2 November 2023	40e6841295a0b3467977edd6663a92a2ce9c7adf	Fixes + Non Inclusion Proofs
4	6 November 2023	d2d4dd7ca6b9b5c681456136c801414b79a285e6	Fixes

For the cairo code, the compiler version `2.2.0` was chosen. At the time of this review (September 2023) Starknet v0.12.2 was live on mainnet. This review cannot account for future changes and possible bugs in Starknet and it's libraries.

The following file was in scope of this review:

```
data_structures/eth_mpt.cairo
data_structures/mmr/mmr.cairo
data_structures/mmr/peaks.cairo
data_structures/mmr/proof.cairo
data_structures/mmr/utils.cairo

encoding/rlp.cairo

hashing/hasher.cairo
hashing/keccak.cairo
hashing/poseidon.cairo

utils/array.cairo
utils/bitwise.cairo
utils/math.cairo
utils/types/byte.cairo
utils/types/words64.cairo
```

This review focussed on CairoLib as used within Herodotus on Starknet. Caution is advised when this library is used in other projects. Reliability and/or the correct usage in other projects is not covered by this review.

2.1.1 Excluded from scope

Any file not listed above is excluded from the scope.

2.2 System Overview

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

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

Herodotus offers CairoLib, a library that implements RLP (Recursive Length Prefix) decoding, wrappers of Keccak256 and Poseidon hash functions, MPT (Merkle Patricia Trie), and MMR (Merkle Mountain Range) algorithms with helper functions in utils.

2.2.1 Customized Type

A customized type `Words64`, as an alias of `Span<u64>`, is defined and used across the library. In most cases of CairoLib, it is expected that the individual `u64` is in little endian, but the overall `Span<u64>` order is big endian.

`Words64` implements the `TryInto` trait to convert itself into a `u256`. `try_into()` reverts in case the input `Words64` is too long to fit within 256 bits. It also reverts in case the input is an empty `Words64`. Otherwise, it converts a span of 64 bit little endian words into a little endian `u256`.

Function `slice_le` is implemented to slice a `Words64` with respect to the individual little-endian and overall big-endian encoding. It always starts from the left word, and reads bytes from right to left within individual words. It will return a new `Words64` which is also encoded in the same way.

2.2.2 RLP Decoding

Recursive Length Prefix (RLP) serialization is used extensively in Ethereum to standardize the transfer of data between nodes in a space-efficient way. The type of the current decoding bytes is determined by the first byte of the input. `rlp_decode()` is implemented in CairoLib, which accepts a `Words64` as input, and decodes it into a single byte, a string, or a list. In case there are trailing garbage bytes in the RLP encoding, they will be discarded. Decoding RLP encoded nested list is not supported. The decoded RLP item as well as the length of decoded bytes will be returned.

2.2.3 Hash Functions

2.2.3.1 Keccak256

A wrapper function `keccak_cairo_words64` is defined, which calls `cairo_keccak()` in the Starknet core library to invoke the system call `keccak_syscall()`. It accepts a list of 64-bit little-endian words (`Words64`) as input. The hash is returned in little endian representation.

2.2.3.2 Poseidon

Poseidon is a family of hash functions designed for being very efficient as algebraic circuits. Starknet's version of Poseidon is a sponge construction based on the Hades permutation over a three element state. `PoseidonHasher` is implemented as a wrapper on top of the `hades_permutation` and `poseidon_hash_span` in the Starknet core library. It exposes `hash_single()`, `hash_double()`, and `hash_many()` to compute the digest of any amount of `felt252`.

2.2.4 Merkle Patricia Trie

MPT (Mekle Patricia Trie) is used in Ethereum as a cryptographically authenticated data structure to store all `(key, value)` bindings. Multiple MPTs are built upon the production of a new block. In the execution block header, the `stateRoot` represents the root of the World State Trie, which provides the bindings from an Ethereum address to its account state. `storageRoot` is a field in the account state representing the root of the Account Storage Trie, which provides the bindings from a storage slot to the value stored. There are Receipts Trie and Transactions Trie as well. One can prove the inclusion of a `(key, value)` in the trie by constructing a merkle proof which encodes all the nodes along the path from the root to the value. The key is a list of nibbles that should match the nibbles stored in the nodes along the way down the proof.

There are four types of node in the Ethereum MPT:

1. NULL Node - represented as the empty string.
2. Branch Node - a 17-item node, which contains hashes of its 16 children and the value stored in this node.
3. Extension Node - a 2-item node, which contains the shared nibbles and the hash of the next node.
4. Leaf Node - a 2-item node, which contains the nibbles and the value stored.

Each nibble is a hexadecimal character represented by 4 bits. Compact encoding is applied to store the nibbles into bytes:

- Extension Node with even nibbles: prepend a `0x00` byte.
- Extension Node with odd nibbles: pack the first nibble `?` with hexadecimal character `1` as: `0x1?`.
- Leaf Node with even nibbles: prepend a `0x20` byte.
- Leaf Node with odd nibbles: pack the first nibble `?` with hexadecimal character `3` as: `0x3?`.

Function `decode_rlp_node` decodes a `Words64` into MPT nodes based on the length of decoded list and the 1 byte prefix of the node. It will revert in case the list length does not fit into any nodes, or the prefix is invalid.

Function `hash_rlp_node` calls `keccak_cairo_words64()` to hash an RLP encoded node. However, as `keccak_cairo_words64()` reviewed in **Version 1** discard trailing zero bytes, it may result in a different hash compared to ethereum `Keccak256`.

Function `verify` accepts a `u256 key`, a `usize key_len`, and a `Span<Words64>` proof (a list of RLP encoded nodes) as input. It traverses the nodes top-down to decode it and verify if the nibbles stored match the next nibbles in the key. In case of a successful verification, it will return the value associated with the key.

2.2.5 Merkle Moutain Range

Merkle Mountain Range is an alternative to Merkle trees. It can be seen as list of perfectly balanced binary trees, where each individual is a simple merkle tree. A Merkle Mountain Range (MMR) is strictly append-only: elements are added from the left to the right, adding a parent as soon as 2 children exist, filling up the range accordingly. The parent is simply a hash of its two children. As a result, the MMR has multiple merkle trees thus a list of peaks.

Contrarily to a Merkle tree, a MMR generally has no single root by construction. Thus, a `bagging` operation is introduced to construct the root of all the peaks together with the size of the tree. If we denote the Poseidon hash function as `H`, the number `i` peaks as `P_i` (from left to right), and the last position as `N`, then the `bagging` operation will be be:

$$root = H(N, H(P_0, H(...H(P_{i-2} H(P_{i-1}, P_i)))))$$

In CairoLib, the struct MMR contains only two fields: `root` and the `last_pos` instead of storing all the peaks. Two main functions exposed are:

1. `append()` - to append a new hashed element to the MMR, the user needs to submit the correct peaks of the current MMR which will be validated. The last position and new MMR root will be updated accordingly.
2. `verify_proof()` - one can verify the inclusion of a hashed element by providing a merkle proof and the correct peaks. It will computes the merkle root based on the proof and verify if the constructed merkle root is a member of the validated peaks.

2.2.6 Roles and Trust Model

This library is expected to be mainly used by Herodotus, however, we assume other users of this library to carefully follow the specifications and fully understand the caveats.

2.2.7 Changes in Version 3

The following functional changes were introduced in Version 3:

- eth_mpt.cairo: `verify()` has been updated to support non-inclusion proofs.

3 Limitations and use of report

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

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

4 Terminology

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

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

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

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

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

5 Findings

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

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

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

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

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	1

- Missing Length Validation in MPT Verify Code Corrected

High-Severity Findings	2

- Keccak Discards Leading Zero Bytes in Last Little Endian Words64 Code Corrected
- MMR: Incorrect Root Update Possible, Insufficient Peaks Validation Code Corrected

Medium-Severity Findings	6

- Empty Extension and Leaf Nodes Code Corrected
- Incorrect Break Condition in Lazy Decode Code Corrected
- Keccak: Unsupported Empty Bytes Input Breaks EVM Equivalence Code Corrected
- MMR Verify Proof: Different Nodes Can Use the Same Index Code Corrected
- Missing Boundary Check in MPT Verification Code Corrected
- Words64TryIntoU256LE Does Not Automatically Pad Input Code Corrected

Low-Severity Findings	2

- Bit_Length Will Revert if Input Most Significant Bit Is 1 Code Corrected
- Right Shift Reverts on Bit Length Input Code Corrected

6.1 Missing Length Validation in MPT Verify

Correctness Critical Version 1 Code Corrected

CS-HRDTCL-010

When verifying a MPT proof by `verify(key, key_len, proof)`, the nodes nibbles (denoted as `n-nibbles`) are decoded and matched with the next nibbles in the key (denoted as `k-nibbles`). However, length validation of the remaining nibbles is missing in the logic of matching both the Extension Node as well as the Leaf Node. Let's assume there is an MPT with two valid proofs, each contains two Branch Node, One Extension Node, and One Leaf Node denoted as:

$$proof1 = Branch[b] || Branch[c] || Extension[de] || Leaf[f2]$$
$$proof2 = Branch[a] || Branch[b] || Extension[cd] || Leaf[0e]$$

In the following description, we will use these dummy proofs as examples.

Leaf Node: it retrieves the elements from `k-nibbles` and `n-nibbles` and checks if they are equal one by one. The matching will be regarded as successful if it reaches the end of the input key. However, there could still be some `n-nibbles` left unmatched.

- As `0xbcdef2` is a valid key for proof1, `verify(0xbcdef2, 6, proof1)` will succeed. However, `verify(0xbcdef, 5, proof1)` will also succeed, even though it does not fully traverse the nibbles in the Leaf Node. As a result, a partial key can be verified and the caller will get the same value stored in the Leaf Node.

Extension Node: it retrieves the elements from `k-nibbles` and `n-nibbles` and check if they are equal one by one. The matching will be regarded as successful if it reaches the first nibble that does not match. However, this implies the following unexpected consequences:

- As `n-nibbles` are stored in `Span<u64>` in little endian, the padding zeros are indistinguishable from valid 0 nibbles. In case the next input nibble is 0, it will be accidentally matched with the padding in the Extension Node, leading to a failure when matching the next node. For example, `verify(0xabcd0e, 6, proof2)` will fail even though proof2 is valid.
- An input key that fully skipped the Extension Node nibbles can bypass the matching in the Extension Node and be successfully matched. In contrast to the failure on validating a legitimate key by `verify(0xabcd0e, 6, proof2)`, a forged key `0xab0e` which fully skips the Extension Node will succeed in `verify(0xab0e, 4, proof2)` since it directly matches the Leaf Node.

In summary, the missing length validation between the nibbles remaining and that stored in the Extension Node or the Leaf Node implies:

1. A partial key can be verified by only matching a prefix of nibbles in the Leaf Node.
2. A legitimate key and proof may fail in case a padding 0 in an Extension Node is accidentally matched with the next nibble in the key.
3. A forged key that skips nibbles of an Extension Node can be verified successfully.

Code corrected:

The length of the nibbles stored in the nodes is propagated after RLP decoding and taken into account in MPT verification now:

1. Leaf Node: A partial key no longer works, a check ensures that all nibbles left from the input key have been matched with all the nibbles stored in the leaf node.
2. Extension Node: The representation of the extension node now additionally contains the number of shared nibbles. The matching is now considered successful if all nibbles match and the code moves to the next node. This prevents skipping nibbles of an Extension node (Problem 3) as well as accidentally matching padding zeros in case the next nibble of the key is zero (Problem 2).

6.2 Keccak Discards Leading Zero Bytes in Last Little Endian Words64

Correctness | High | Version 1 | Code Corrected

CS-HRDTCL-002

`fn keccak_cairo_words64(words: Words64)` does not work correctly in case there are trailing zeros in the big-endian input bytes (represented as leading zeros in the last item of little-endian Words64).

For example, the big-endian bytes [0xaaaaaaaaaaaaaabb, 0x653800] would be represented as [0xbbaaaaaaaaaaaaaa, 0x3865] in Words64. There is no way for function `bytes_used_u64` to identify how many leading zeros should be included since it operates on the value. The trailing zeros are an important part of the input however, omitting them will lead to a different hash.

Code corrected:

Function `keccak_cairo_words64` now takes the byte length of the last word (`last_word_bytes`) as an extra input. Using this the hash can be computed correctly even in case of leading zero bytes.

6.3 MMR: Incorrect Root Update Possible, Insufficient Peaks Validation

Security High Version 1 Code Corrected

CS-HRDTCL-007

A Merkle Mountain Range struct consists of:

1. `last_pos`: the last position (size) of the elements stored. The shape of the Merkle Mountain Range as well as the number of peaks can be deterministically computed from the `last_pos`.
2. `root`: a hash consists of all peaks and the `last_pos`.

In `mmr.append()`, `peaks.valid()` does not check if the length of the input peaks is correct with respect to `last_pos`. Given the special construction in the bagging algorithm shown below, one can submit left side peaks together with an intermediate Poseidon hash of all right side peaks, which will also pass the peaks validation.

$$root = H(N, H(P_0, H(\ldots H(P_{i-2} H(P_{i-1}, P_i)))))$$

For example, if we assume currently there are 4 peaks denoted as `[P_0, P_1, P_2, P_3]`. One can submit the forged 3 peaks as `[P_0, P_1, H(P_2, P_3)]` to bypass the peaks validation. Then `mmr.append()` will compute the updated peaks and new root according to the wrong former peaks, and the shape of the MMR will be disrupted.

In summary, one can update the root and peaks to wrong values by replacing some right side peaks with intermediate bagging results. The insufficient peaks validation (`peaks.valid()`) also appears in `mmr.verify_proof()`.

Code partially corrected:

In Version 2, `append()` has been updated to ensure the amount of peaks given as input is correct given the `last_pos` of the current mmr.

In addition, `verify_proof()` now ensures the length of the given proof matches the expected height of the peak. This however is insufficient and still allows verification against a forged peak (see issue MMR Verify Proof: Different Nodes Can use the same index).

Code corrected:

In Version 3 `verify_proof()` now features the same check as `append()` to ensure the amount of peaks given as input is correct.

6.4 Empty Extension and Leaf Nodes

Correctness Medium Version 3 Code Corrected

CS-HRDTCL-008

In Version 3 of Cairo-lib, `mpt.verify()` has been changed to support:

1. When encountering an extension node that contains no nibbles, the function will continue its process with the subsequent node.
2. In the case where a leaf node has no nibbles and the provided key has been completely processed, the function will immediately return the node's value.

The Ethereum Yellowpaper states the following on page 21:

```
Leaf: A two-item structure whose first item corresponds to the nibbles in the
key not already accounted for by the accumulation of keys and branches traversed
from the root. The hex-prefix encoding method is used and the second parameter
to the function is required to be 1.

Extension: A two-item structure whose first item corresponds to a series of nibbles
of size greater than one that are shared by at least two distinct keys past the
accumulation of the keys of nibbles and the keys of branches as traversed from the
root. The hex-prefix encoding method is used and the second parameter to the function is
required to be 0.
```

https://ethereum.github.io/yellowpaper/paper.pdf

These would be 'empty` nodes which don't exist.

Code corrected:

The respective conditional branches for empty nodes have been removed.

6.5 Incorrect Break Condition in Lazy Decode

Correctness | Medium | Version 3 | Code Corrected

CS-HRDTCL-004

An RLP encoded list (with total length `rlp_byte_len = p_len + len`) consists of:

1. A prefix that reveals the list type and length of its content, whose length is denoted as `p_len`.
2. The actual content of the list, whose length is denoted as `len`.

`rlp_decode_list_lazy()` will decode only the corresponding indices of an RLP encoded list by reading through the contents of the RLP encoded list and only slice its content if it is a target. However, one of its break condition (check if it reaches the end of the RLP encoding) compares the current cursor (`current_input_index`) with the partial `len`. As a result, `rlp_decode_list_lazy()` may return with an error due to reading out of bounds even though it hasn't.

Code corrected:

The break condition has been corrected with `rlp_byte_len`.

6.6 Keccak: Unsupported Empty Bytes Input Breaks EVM Equivalence

Correctness | Medium | Version 1 | Code Corrected

CS-HRDTCL-012

`fn keccak_cairo_words64(words: Words64)` does not support empty bytes as input, hence is not equivalent to the EVM's keccak256 opcode.

In Version 2: Empty bytes have been treated as a special hardcoded case, however the hash of empty bytes returned is in big endian. For all other inputs, this function returns the hash calculated by `cairo_keccak()`, which returns the hash in little endian representation.

Code corrected:

The function now returns the correct value for the empty input in little endian. The description of the function has been enhanced to clarify the format of the return value / difference to the EVM opcode.

6.7 MMR Verify Proof: Different Nodes Can Use the Same Index

Design Medium Version 1 Code Corrected

CS-HRDTCL-006

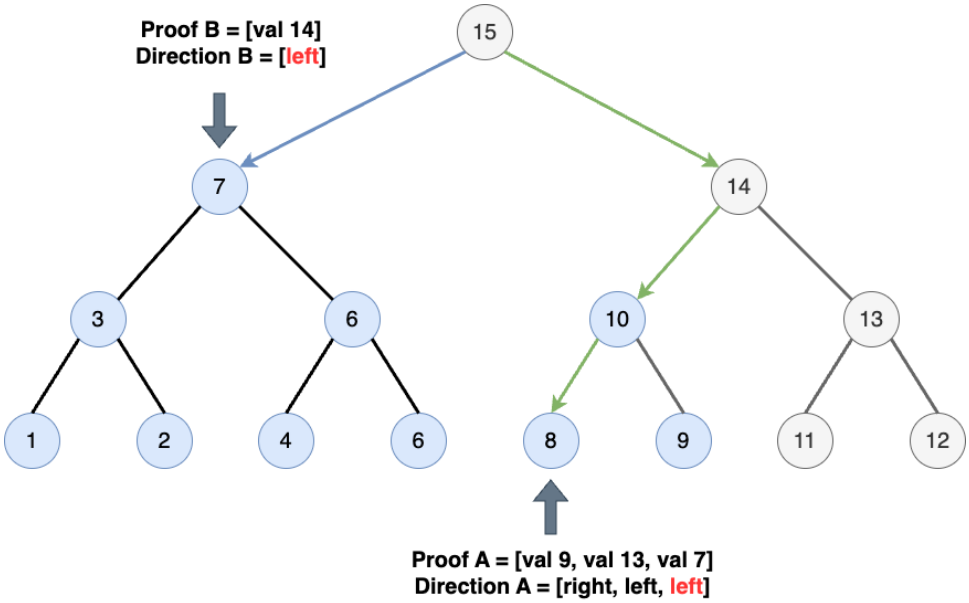
In MMR `verify_proof()`, the user needs to submit:

- The index of the hash element to build a direction array, that determine the order (left or right) of hash when reconstructing the merkle root.
- A hash of the to be verified node.
- A merkle proof array that reconstruct the root of a merkle trie that contains the node.
- MMR Peaks to check if the reconstructed root is one of the peaks.

After it has been ensured that the peaks are valid, the peak based on the given proof, index and hash is calculated:

```
let peak = proof.compute_peak(index, hash);
```

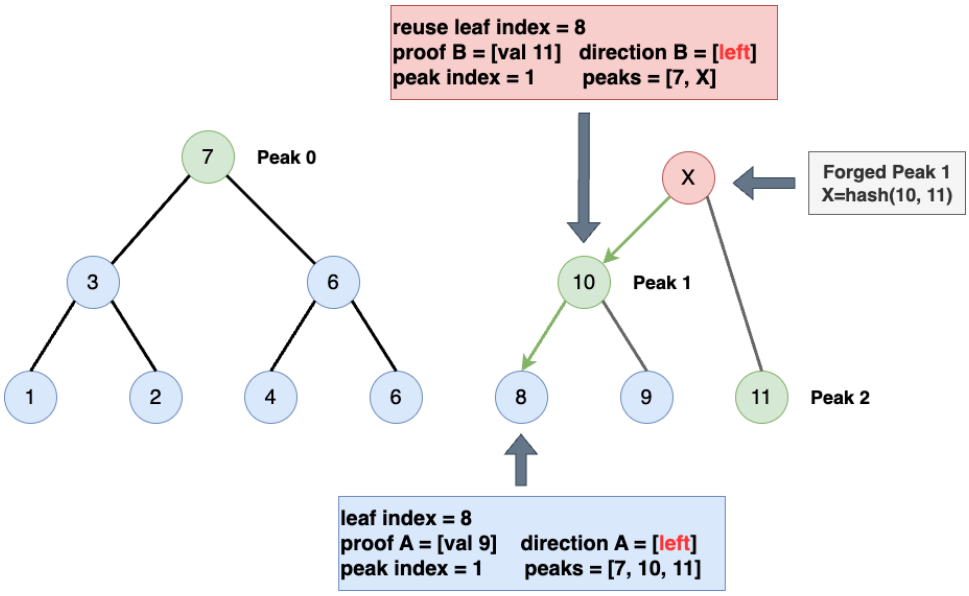
However, the reconstruction ends if we have run out of the merkle proof elements regardless of the direction array. As a result, one index of a node can also be used in the proof of another node in case the shared part is the same.



In the figure above, blue nodes denote the real MMR and the grey nodes are imaginary for building direction array. Knowing that the elements of direction array is consumed in a reverse order, it is obvious that `(Poof B, Direction A)` can also be used to verify against node 7.

Note this also enables the verification against an intermediate node instead of a leaf node. It may not be a legitimate use case in practice, and external systems using this library should execute caution to avoid this being abused.

The updated code of Version 2 ensures the provided proof has the correct length and compares the computed peak of the element to the specific peak in the list of peaks. The list of peaks however is an input to the function and hence may be manipulated. Although now very restricted, it's still possible to verify against a forged peak as illustrated in the following example:



Code corrected:

In Version 3 `verify_proof()` has been strengthened: A check now ensures the number of peaks given as input is correct given the `last_pos` of the current MMR. Since MMRs are deterministic, this together with the validations of the peaks (`peaks.valid()`) ensures `verify_proof()` verifies against the correct MMR without forged peaks.

6.8 Missing Boundary Check in MPT Verification

Correctness Medium Version 1 Code Corrected

CS-HRDTCL-011

Note: This issue was discovered by Herodotus.

When matching the input nibbles with the nibbles stored in an extension node or a leaf node (in `MPT.verify()`), the nibbles stored in the nodes, represented as individually little endian and overall big endian `Words64`, are read in a special order:

- Within one `u64`, bytes are loaded from right to left.
- Within one byte, nibbles are loaded from left to right.
- Once it finishes an `u64`, it will jump to the next u64.

However, the jump in step 3 does not check if it has reached the end of the `Words64`. Consequently, `MPT.verify()` may revert on a valid proof due to reading out of bounds or overflow.

Code corrected:

Herodotus has updated the logic of the code to avoid reading out of bounds or overflow.

6.9 Words64TryIntoU256LE Does Not Automatically Pad Input

Correctness Medium Version 1 Code Corrected

CS-HRDTCL-005

`Words64TryIntoU256LE` implements `try_into()` to convert a overall big-endian and individually little-endian `Words64` (namely `Span<u64>`) into a little-endian `u256`. In case the input length is less than `4 u64`, it will not automatically add leading zero padding upon conversion, and the result may be incorrect. For example:

1. Assume the original big-endian `u256` is 0x11aaaaaaaaaaaff.
2. If its Words64 representation is an array [0xffaaaaaaaaaaaa11] with length 1.
3. As `Words64TryIntoU256LE` does not pad the input array, the result little-endian `u256` will be 0xffaaaaaaaaaaaa11.
4. If one converts it back to big-endian, one will eventually get a much bigger value: 0x11aaaaaaaaaaaff00000000000000000000000000000000000000000000000000.

Code corrected:

`Words64TryIntoU256LE` has been removed and the following functions have been added to facilitate the conversion between endianess with different bytes length:

```
fn as_u256_le(self: Words64, bytes_used: usize) -> Result<u256, felt252>;
fn as_u256_be(self: Words64, bytes_used: usize) -> Result<u256, felt252>;
```

6.10 Bit_Length Will Revert if Input Most Significant Bit Is 1

Correctness Low Version 1 Code Corrected

CS-HRDTCL-013

`bitwise::bit_length()` computes the result by finding the first n that `2^n` is larger than the input number. Consequently, if the input number of type `T` has `1` on the most significant bit, and type `T` implements overflow protections, `bit_length()` will revert due to `2^n` overflows type `T`. For example, to measure the bit length used for `128_u8`, it will compute `2^8` in the last round and revert by overflowing `u8`.

Code corrected:

The code has been corrected to measure the bit length by iteratively dividing the input by 2.

6.11 Right Shift Reverts on Bit Length Input

Design | Low | Version 1 | Code Corrected

CS-HRDTCL-014

`bitwise::right_shift(num, shift)` will divide the number by two to the power of shift (computed by `bitwise::pow()`). `pow()` accepts the input of type `T` and returns the result in the same type `T`. Consequently, `right_shift()` will revert in case `shift >= bit_length(max(T))` and type `T` implements overflow protection, since the return value of `pow()` overflows type `T` . For example, in case the input number is of type `u32`, `right_shift()` will revert if `shift==32` since `2^32>max(u32)`.

Code corrected:

Code has been changed to achieve right shift by dividing input number by 2 iteratively instead of computing `pow()`.

6.12 Fast Power May Improve Gas Efficiency

Informational | Version 1 | Code Corrected

CS-HRDTCL-015

The current implementation of `bitwise::pow()` has a time complexity `O(n)`. A fast power implementation only requires `O(log(n))` time complexity and may improve gas efficiency.

Code corrected:

A fast power (`fast_pow()`) implementation has been added. In addition, `bitwise::pow()` has been changed to execute `slow_pow()` if the exponential is below 16, otherwise, it will execute `fast_pow()`.

6.13 Order of Evaluation Can Be Enforced

Informational | Version 1 | Code Corrected

CS-HRDTCL-009

At the time of this review, there is no Starknet official documentation of the order of expressions evaluation. In `words64::Words64TryIntoU256LE`, the execution order of the following code is not guaranteed to be always from left to right. Explicit brackets can be used to enforce the order of evaluation.

Code corrected:

Explicit brackets have been added in `words64::Words64TryIntoU256LE` to enforce the evaluation order.

6.14 Unused Import

Informational | Version 1 | Code Corrected

CS-HRDTCL-001

`words64.cairo` imports `bitwise::right_shift`, however, it is never used.

Code corrected:

The unused import has been removed.

6.15 Words64TryIntoU256LE Reverts in Case of Empty Input

Informational | Version 1 | Code Corrected

CS-HRDTCL-003

`Words64TryIntoU256LE` implements `try_into()` to convert a `Words64` (namely `Span<u64>`) into a `u256`. In case the input exceeds 256 bits, it will return an `Option::None`. Moreover, it does not accept empty span as input, and will reverts in this case.

In Version 2, `Words64TryIntoU256LE` has been adjusted to return 0 in case the input is empty.

Code corrected:

In Version 3, two functions (`as_u256_le` and `as_u256_be`) facilitate the conversion between different endianness for variable input length.

7 Notes

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

7.1 Lazy Decode Returns Items With Ascending Indices

Note Version 3

`rlp_decode_list_lazy(input: Words64, lazy: Span<usize>)` will iterate the items in the input (RLP encoded list) and append the item to the output array in case the current index is in the required indices list (`lazy`). As a result:

- Duplicated indices in (`lazy`) will only be counted once.
- In the output array, the items will be in an ascending order according to their indices in the input array, and **irregardless of** their positions in the input array (`lazy`).

7.2 Left Shift May Revert Due to Overflow

Note Version 1

`bitwise::left_shift()` computes the result by multiplying the input of type `T` with two to the power of shift. In case type `T` implements overflow protection for trait `TMul`, the result must also fits within type `T`, otherwise it will revert due to overflow.

7.3 MPT Verify Can Not Get the Value in the Root Branch Node

Note Version 1

A branch node can store 16 hashes of the children nodes and one value. In case the root node is a branch node, theoretically we should be able to retrieve the value stored inside. Whereas `mpt.verify()` does not support this and would revert due to underflow when computing `key_pow2`. Though this shouldn't be a practical use case in Ethereum.

7.4 Missing Length Information of Type Words64

Note Version 1

A customized type `Words64`, as an alias of `Span<u64>`, is defined and heavily used across the library. In most cases, it is used to store bytes, however, the length of bytes within each `u64` is missing. As a result, the leading valid zeros bytes in `u64` is indistinguishable from padding in `bytes_used_u64`. This has lead to various problems e.g. in `keccak256` and MPT verification.

Within CairoLib and Herodotus on Starknet the project this library has been build for, length information of data stored in `Words64` is handled separately throughout the codebase. Other users of the library must be aware and handle this correctly.

7.5 RLP Decode Will Discard Garbage Bytes

Note Version 1

`rlp_decode()` will only read and decode bytes according to the length that is encoded in the input prefix. In case there are trailing garbage bytes, they will be discarded and `rlp_decode()` will not revert.

7.6 Reverse Endianness U64 Reverts if Significant Bytes Are Larger Than 8

Note Version 1

`reverse_endianness_u64()` will reverse the endianness of a `u64` given the significant bytes the user wants to reverse. There is no restrictions on the significant bytes, in case the significant bytes are larger than 8, `pow2()` will return 0 in the last iteration of the loop and `reverse_endianness_u64()` will revert due to division by zero.