Security Review Report
NM-0069 Polygon Id

NETHERMIND

(Apr 18, 2023)

Contents

1 Executive Summary

This document presents the security review performed by Nethermind in the Polygon Id Smart Contracts. The Polygon Id is a decentralized and permissionless identity framework for web2 and web3 applications based on the principles of Self-Sovereign Identity (SSI) and cryptography. With the help of zero-knowledge proofs, users can prove their identity without exposing their private information. The audit focuses on Iden3 smart contracts. Iden3 is an open-source protocol that provides the foundations for Polygon Id. The protocol defines how the parties communicate and interact. Polygon Id is an abstraction layer to enable developers to build applications leveraging the Iden3 protocol. **The audited code consists of** 687 lines of Solidity with code coverage of 94%. The Polygon team provided two documents to assist the audit presenting an overview of the contracts and how to run the test suite. The audit was supported by the documentation accessible from the Polygon ID wiki.

The audit was conducted using (a) manual analysis of the codebase, (b) automated analysis tools, (c) simulation of the smart contracts, and (d) creation of test cases. The Nethermind Formal Verification and Cryptography Research teams also provided support for the audit. The audit was supplemented with multiple testing techniques, such as black-box testing, white-box testing, and property testing. The white-box tests were enhanced by dynamic analysis to uncover untested code branches.

In the first re-audit, we identified 16 points of attention, of which two were classified as `Low` severity, while the remaining 14 points of attention were classified as either `Informational` or `Best Practices`. Following the re-audit, the Polygon team addressed 12 issues, while four issues were acknowledged.

After the audit, the Polygon team decided to refactor the implementation to remove the restriction of only having one unique state per contract, which could potentially limit the future evolution of the system. Our team reviewed these changes in re-audit 2. **In the second re-audit, we identified five points of attention**, of which two were classified as `Informational` and three were classified as `Best Practices`. The Polygon team addressed four issues, while one issue was acknowledged. The acknowledged issue pertains to our request for the creation of a test case to evaluate the logic for updating implementations, which is a standard procedure that the Polygon team has expertise in. **Fig. 1 summarizes all the issues reported in re-audits 1 and 2**.

This document is organized as follows. Section 2 presents the files in the scope of this audit. Section 3 presents the assumptions for this audit. Section 4 summarizes the issues. Section 5 presents the system overview. Section 6 discusses the probability of collision when the system receives more identities (users). Section 7 presents an abstract formal specification of the application. Section 8 discusses the risk rating methodology adopted for this audit. Section 9 details the issues raised in audit 1. Section 10 details the issues raised in audit 2 (after the audit, the Polygon team decided to refactor the implementation. The refactoring is intended to remove the limitation of only one unique state per contract). Section 11 presents the complementary validations performed by Nethermind for testing the application. Section 12 discusses the documentation provided by the client for this audit. Section 13 presents the compilation, tests, coverage, and automated tests. Section 14 concludes the document.

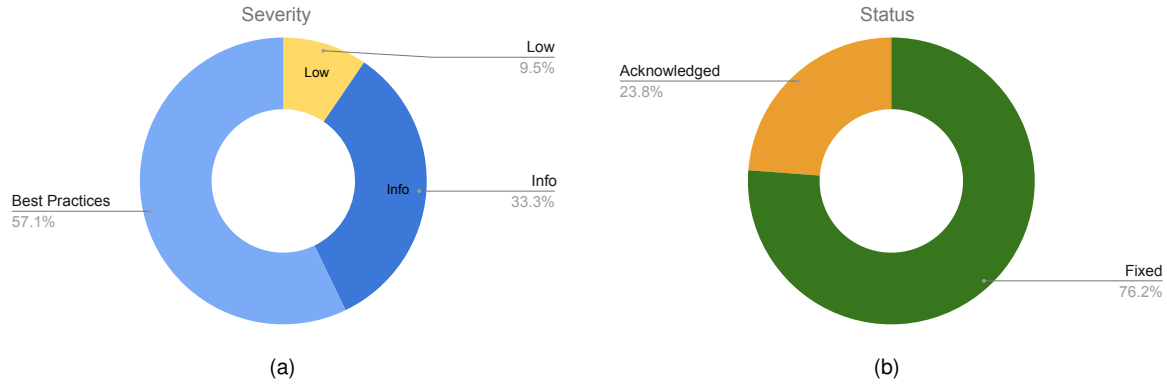


Fig. 1: Distribution of issues: Critical (0), **High** (0), **Medium** (0), **Low** (2), **Undetermined** (0), **Informational** (7), **Best Practices** (12). **Distribution of status: Fixed** (16), **Acknowledged** (5), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Jan. 31, 2023
Response from Client	Feb. 15, 2023
Final Report Reaudit 1	Mar. 16, 2023
Final Report Reaudit 2	Apr. 18, 2023
Methods	Manual Review, Automated Analysis
Repository	Polygon ID
Commit Hash (Initial Audit)	9c8f5d6132b8918b2889ab4c8bda39d87d06e312
Commit Hashes (Reaudit)	d9be60d7c92d331058135f4fa124969fb02fdcf3 80398a1b46c4108ee9dcccb710486e7a3bc5ec99 ebe466957f4d8af8e2c11ad5249e3ac219b9145d c9c661ef00811b8f1eb9367519d4f230aa1b2842 347dd01048110f8da49069835a93a6fe0bd389c0 3245a7563ecc3f8876682b67c965437ce888ca62 a04690649188b71cceddc2ef81fcbec54926b7aa 278e292d1cd7f6063f5ebeb0ef855d00f51af528 6ac5bca7fcd2138f639bcf39722107c0b4e37b9a
Documentation	README
Documentation Assessment	High
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	contracts/lib/Smt.sol	427	127	29.7%	60	614
2	contracts/state/StateV2.sol	260	173	66.5%	44	477
	Total	**687**	**300**	**43.6%**	**104**	**1091**

3 Assumptions

The prepared security review is based on the following assumptions:

- The off-chain code which interacts with the on-chain contracts is safe;
- The `Poseidon` hash library is implemented correctly;
- The circuits of the zero-knowledge proof are correct;
- The `verifier` of zero-knowledge proof is responsible for checking the validity of the genesis state for a given index;
- The `verifier` of the zero-knowledge proof is responsible for checking the validity of state transitions from the old state to the new one for a given index;
- The sparse Merkle Tree implementation can hold up to 2^{31} leaves.

4 Summary of Issues

Audit 1

	Finding	Severity	Update
1	Lack of a two-step process for transferring ownership	Low	Fixed
2	Unnecessary space allocation in `Proof.siblings`	Low	Fixed
3	Code and specification not matching	Info	Fixed
4	Nodes with incorrect depth are allowed	Info	Fixed
5	Owner can change the verification logic after the contract's deployment	Info	Acknowledged
6	Privileged Roles and Ownership	Info	Acknowledged
7	Upgradability	Info	Acknowledged
8	Avoidable reversion in function `getRootHistory(...)`	Best Practices	Fixed
9	Functions that can have `external` visibility instead of `public`	Best Practices	Fixed
10	Memory variables should be initialized	Best Practices	Fixed
11	Not checking the verifier contract for `address(0x0)`	Best Practices	Acknowledged
12	Redundant input arguments in function `_pushLeaf(...)`	Best Practices	Fixed
13	Special values able to be used as normal input	Best Practices	Fixed
14	Unnecessary path specification in import	Best Practices	Fixed
15	`abicoder v2` pragma is not needed since version `0.8.0`	Best Practices	Fixed
16	Variable can be `uint256` instead of `uint64`	Best Practices	Fixed

Audit 2

	Finding	Severity	Update
1	Inconsistent behavior of getter function for zero root	Info	Fixed
2	NatSpec comment missing in function `calculateBounds(...)`	Info	Fixed
3	Gaps with round numbers	Best Practices	Fixed
4	Not testing implementation updates	Best Practices	Acknowledged
5	Uninitialized proxy implementation	Best Practices	Fixed

5 System Overview

The Polygon ID, with the help of zero-knowledge proofs, lets users prove their identity without exposing their private information. This section was written based on the codebase audited in audit 1, before the refactoring performed by the Polygon team. **The audit is based on two contracts:** a) *StateV2.sol*; and b) *Smt.sol*. The contract *StateV2.sol* stores meta information about identities. The contract is *OwnableUpgradeable*, which means that it uses the `EIP-1967` proxy pattern for holding the state information, where the proxy contract store the address of the logic contract they delegate to, as well as other proxy-specific information. The contract `Smt.sol` implements a Sparse Merkle Tree used by the *StateV2.sol* for managing its data. The contracts are described below in accordance with the structural diagram presented in Fig.2.

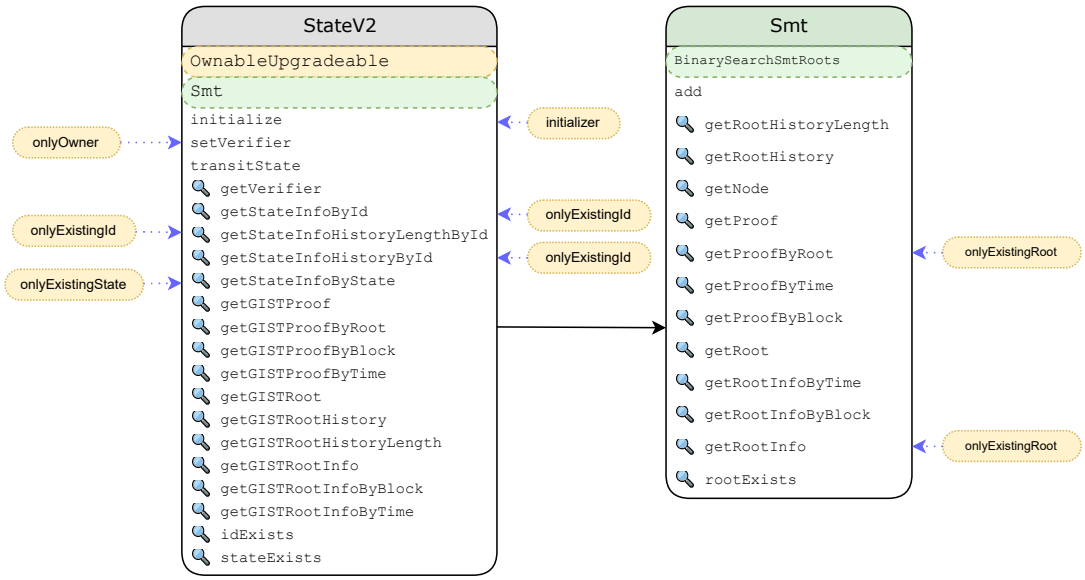


Fig. 2: Structural Diagram of the Contract

5.1 `StateV2.sol`

The contract `StateV2.sol` uses three structs: `StateData`, `StateEntry`, and `StateInfo`. The struct `StateEntry` holds the metadata of each identity state. The struct is reproduced below.

```
struct StateEntry {
    uint256 id;
    uint256 timestamp;
    uint256 block;
    uint256 replacedBy;
}
```

The struct `StateData` stores all the state data. In this struct, the field `statesHistories` holds the history per each identity, while the field `stateEntries` stores data related to the state of the identity.

```
struct StateData {
    mapping(uint256 => uint256[]) statesHistories;
    mapping(uint256 => StateEntry) stateEntries;
}
```

The struct `StateInfo` is used for public interfaces to represent state information. This structure holds information about identity identifiers when the state was committed to the Blockchain, the block number when the state was committed, and the state which replaced this state for the identity. The struct is reproduced below.

```
struct StateInfo {
    uint256 id;
    uint256 state;
    uint256 replacedByState;
    uint256 createdAtTimestamp;
    uint256 replacedAtTimestamp;
    uint256 createdAtBlock;
    uint256 replacedAtBlock;
}
```

Thus, for each identity, the contract stores the identity state and its history. The contract also uses a sparse Merkle Tree to store the whole tree (and history) on-chain. This contract has the following public functions.

- `setVerifier()`
- `transitState()`
- `renounceOwnership()`
- `transferOwnership()`

In addition to these functions, the contract provides several `getters` not described here. These `getters` allow to retrieve data such as the `state`, `root` information, check the existence of `ids`, `states`, retrieve the `verifier` address, among others. Fig. 2 lists all the functions implemented in the contract `StateV2.sol`.

5.2 `Smt.sol`

`Smt` is a library implementing a Sparse Merkle Tree (SMT). This implementation keeps the tree history as long as roots are never duplicated. Each leaf of the tree consists of a tuple $(index, value)$, where the `index` represents the identity (and also the location in the tree), while value represents the value assigned to the given id. A Sparse Merkle Tree is an authenticated data structure based on a perfect Merkle tree of intractable size. It contains a distinct leaf for every possible output from a cryptographic hash function and can be simulated efficiently because the tree is sparse (i.e., most leaves are empty).

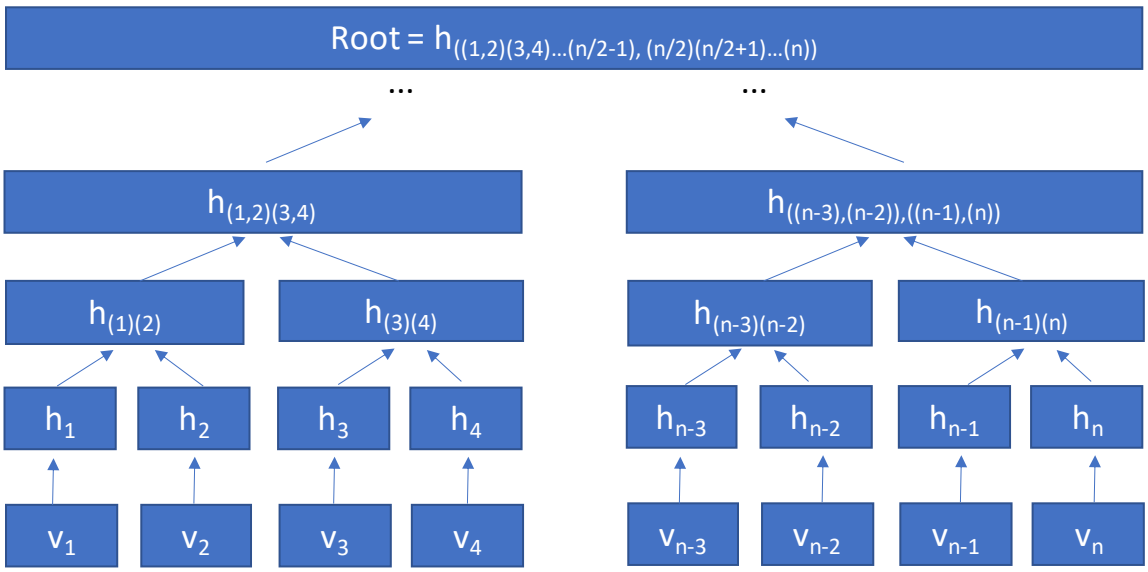


Fig. 3: Representation of a Sparse Merkle Tree

Given a set of values $V = \{v_1, v_2, v_3, \ldots, v_n\}$, the Merkle tree can be computed by generating the set of hashes $H = \{h_1, h_2, h_3, \ldots, h_n\}$ using a cryptographic function over each element of V, where $h_i = hash(v_i)$ and $1 \leq i \leq n$. After computing each $h \in H$, we must generate the second level of the tree (H') by hashing siblings elements, where each element belongs to a single pair. So, $h'_1 = hash(h_1, h_2)$, $h'_2 = hash(h_3, h_4)$, and so forth. By doing so, $|H'| = |H|/2$, i.e., we reduce the cardinality of the set H to its half. Then, we make $H = H'$. We repeat the process until $|H| = 1$, i.e., only one element is left in H. That element is called the root of the Merkle tree. Whenever we have an odd number of elements in the set H, we repeat the last element to always have an even number of elements at any given level.

When considering the implementation being audited, the value v is represented by the tuple $(index, value)$. It is also important to mention that the `index` indicates the item's location in the tree. To locate the key-value pair, the `index` is read bit-by-bit from the right-most bit to the left-most bit while traversing the tree from the root downwards. Bit zero means "follow the edge going to the left", while bit one means "follow the edge going to the right". The main goal of the SMT implementation is to provide inclusion and non-inclusion proofs of identities states into the SMT root. The Smt library contract is used for the state contract (`StateV2.sol`). The Sparse Merkle Tree is defined to have a maximum depth of 32 levels. Since the first depth is zero, the tree can hold up to 2^{31} leaves.

```
uint256 public constant MAX_SMT_DEPTH = 32;
```

The implementation considers three node types. The type `empty` represents a new node that has not been filled with any data yet. The type `leaf` represents a tree leaf, while the type `middle` represents non-leaf nodes. The definition of nodes is presented below.

```
enum NodeType {
    EMPTY,
    LEAF,
    MIDDLE
}
```

The Sparse Merkle Tree data is represented in the struct `smtData` having the mapping of `nodes` (position in the Merkle tree and nodes data) and the history of `roots`. The struct is reproduced below.

```
struct SmtData {
    mapping(uint256 => Node) nodes;
    uint256[] rootHistory;
    mapping(uint256 => RootEntry) rootEntries;
}
```

The struct `Node` holds the type of the node (empty, leaf, middle), the `left` and `right` children, the `index` of the node (location in the tree), and the `value`.

```
struct Node {
    NodeType nodeType;
    uint256 childLeft;
    uint256 childRight;
    uint256 index;
    uint256 value;
}
```

The struct `RootEntry` stores data related to the root of the Sparse Merkle Tree. This struct is used internally, and it has the following fields: `replacedByRoot` (indicates which root has replaced this one), `createdAtTimestamp` (indicates the creation time), and `createdAtBlock` (indicates the creation block). The struct is reproduced below.

```
struct RootEntry {
    uint256 replacedByRoot;
    uint256 createdAtTimestamp;
    uint256 createdAtBlock;
}
```

The code also has the struct `RootInfo` used as an interface to external calls. This struct is very similar to `RootEntry`, with the addition of the following fields `root` (indicates the root of the tree), `replacedAtTimestamp` (indicates the time where the tree has changed), `replacedAtBlock` (indicates the block where the root has changed). The struct is presented below.

```
struct RootInfo {
    uint256 root;
    uint256 replacedByRoot;
    uint256 createdAtTimestamp;
    uint256 replacedAtTimestamp;
    uint256 createdAtBlock;
    uint256 replacedAtBlock;
}
```

Finally, the contract also uses one struct to hold proof-related data. The struct has the `root`, `existence` (indicates if the index exists in the tree), `siblings` (holds the siblings for the proof, `index` (index in the tree), `value` (value stored in the tree), `auxExistence` (indicates that the searched node does not exist in the tree but the auxiliary node was found instead), `auxIndex` (contains the index of an auxiliary node), `auxValue` (contains the value of an auxiliary node). The struct is also reproduced below.

```
struct Proof {
    uint256 root;
    bool existence;
    uint256[MAX_SMT_DEPTH] siblings;
    uint256 index;
    uint256 value;
    bool auxExistence;
    uint256 auxIndex;
    uint256 auxValue;
}
```

5.3 Library `BinarySearchSmtRoots`

The code uses Binary Search to find `roots` of the stored Sparse Merkle Trees in $O(\log_2^n)$. The code can execute the search based on the `block.timestamp` or the `block.number`. Thus, the library `BinarySearchSmtRoots` is applied to the `SmtData`.

```
using BinarySearchSmtRoots for SmtData;
```

```
function binarySearchUint256( Smt.SmtData storage self, uint256 value, SearchType searchType ) ... returns (uint256) {
    if (self.rootHistory.length == 0) { return 0; }

    uint256 min = 0;
    uint256 max = self.rootHistory.length - 1;
    uint256 mid;
    uint256 midRoot;

    while (min <= max) {
        ////////////////////////////////////////////////////////////////////////
        // @audit Compute the index in the middle of the array.
        ////////////////////////////////////////////////////////////////////////
        mid = (max + min) / 2;
        midRoot = self.rootHistory[mid];

        ////////////////////////////////////////////////////////////////////////
        // @audit Define if searching for "block.timestamp" or "block.number".
        ////////////////////////////////////////////////////////////////////////
        uint256 midValue = fieldSelector( self.rootEntries[midRoot], searchType );
        if (midValue == value) {
            ////////////////////////////////////////////////////////////////////////
            // @audit If the elements have been found, start the linear search.
            ////////////////////////////////////////////////////////////////////////
            while (mid < self.rootHistory.length - 1) {
                uint256 nextRoot = self.rootHistory[mid + 1];
                uint256 nextValue = fieldSelector( self.rootEntries[nextRoot], searchType );

                if (nextValue == value) {
                    ////////////////////////////////////////////////////////////////////////
                    // @audit If the next element is equal to the actual element, move forward.
                    ////////////////////////////////////////////////////////////////////////
                    mid++;
                    midRoot = nextRoot;
                }
                else {
                    ////////////////////////////////////////////////////////////////////////
                    // @audit Next element is different. Finish the search.
                    ////////////////////////////////////////////////////////////////////////
                    return midRoot;
                }
            }
            return midRoot;
        }
        else if (value > midValue)            { min = mid + 1; }
        else if (value < midValue && mid > 0) { max = mid - 1; }
        else { return 0; }
}
```

In the contract `Smt.sol`, the Binary Search is called in functions `getRootInfoByTime(...)` and `getRootInfoByBlock(...)`. The Binary Search is implemented using an interactive approach (instead of a recursive one) followed by a Linear Search to find the last element meeting the search criteria. In the extreme case when a large portion of the elements are added to the sparse Merkle Tree in the same `block.timestamp` or `block.number`, the Binary Search can become linear, i.e., the order of complexity can degenerate from $O(\log_2^n)$ to

$O(n)$. However, this is very unlikely to happen, and most of the searches must be completed in $O(\log_2^n) + O(\delta)$, where δ represents the number of equal elements meeting the search criteria. The Binary Search with audit comments is shown above.

To reuse the same implementation, the type of the search is specified using the `enum SearchType` described below.

```
enum SearchType {
    TIMESTAMP,
    BLOCK
}
```

5.4 Roles

The `STATEV2_OWNER` is the address that deployed the contract `StateV2`. The role can:

- Set a new ZKP verifier contract address by calling the function `setVerifier(address newVerifierAddr);`
- Renounce ownership by calling the function `renounceOwnership()`. This leaves the contract without an owner;
- Transfer ownership of the contract to a new owner by calling the function `transferOwnership()`.

6 Analysis of the Probability of Collision for Identities

This section was prepared by **Nethermind's Research Team** for discussing the identity's collision probability. Since identities are stored in a sparse Merkle Tree (SMT) having 32 levels, the sparse Merkle Tree can store up to 2^{31} leaves. In this section, we approach this problem using two complementary approaches. **Initially, we use formal methods to derive the equation that relates** the collision probability to the number of elements stored in the sparse Merkle Tree considering the sparse Merkle Tree's depth. This is achieved by mapping the collision problem into the classical birthday problem. **After that, we approach the collision problem by** instantiating sparse Merkle Trees, adding random elements, and measuring the number of collisions according to the number of elements added.

6.1 Approach: The birthday problem

We would like to include a note on the likelihood of seeing leaf "collisions" that impede an element being added to the Merkle tree in light of the depth limit parameter of 32. With leaves indexed by 256-bit hashes, we have 2^{256} different elements to consider, but a maximum of 2^{31} elements that can fit in the last level of the tree. We, therefore, ask: what is the probability of two leaves, generated at random, occupying the same space in the last level of the SMT? (We call this a *collision*). And how concerning is this probability in ordinary use? **Since the SMT allocates leaves in the tree according only to their least significant 31 bits**, we can consider leaves ℓ_i to be 31-bit sequences, without loss of generality—thus restricting the total options down to $T = 2^{31}$. Then we can phrase our question as follows:

Consider a set of n leaves $\{\ell_1, \ell_2, \ldots \ell_n\}$ which have been chosen uniformly at random. Compute the probability $P_{col} = P(\ell_i = \ell_j,\ i \neq j)$ as a function of n.

This is an instance of a well-known probability problem, the **birthday problem**. The solution (which can be found via combinatorics as in, for example, https://mathworld.wolfram.com/BirthdayProblem.html), is:

$$P_{\text{col}} = 1 - \frac{T!}{(T-n)!T^n} \tag{1}$$

Since the factorials above complicate visualizing the equation, the following estimate is commonly used:

$$P_{\text{col}} \approx 1 - e^{-n(n-1)/2T} \tag{2}$$

Let us graph this estimate below.

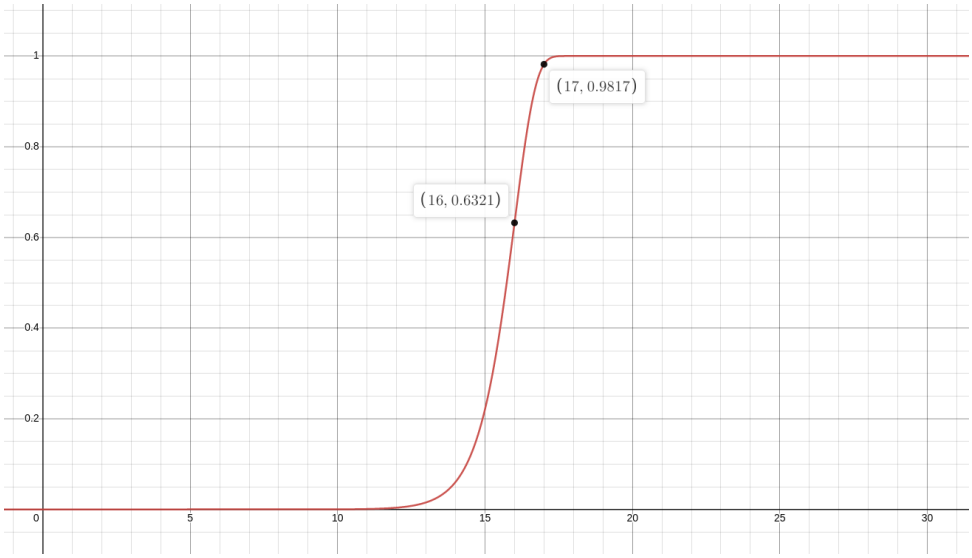


Fig. 4: Estimate of the probability of collision P_{col} as a function of $\log_2 n$. The highlighted points show that $P_{\text{col}} \approx 63\%$ when $n = 2^{16} = 65536$ and $P_{\text{col}} \approx 98\%$ when $n = 2^{17} = 131072$.

6.1.1 Asking the inverse question

For security purposes (and for grasping the meaning of the formula above), it might be more meaningful to frame the question inversely:

What is the value of N required in order for the probability P_{col} to rise to an appreciable number?

Fig. 4 above answers our questions. For $n = 2^{16}$ leaves, seeing a collision is likely, at roughly 63% probability. For $n = 2^{17}$ leaves and beyond, a collision is practically guaranteed. **The fact that a collision is so likely at 2^{17} leaves may seem counterintuitive, especially since it represents only 0.006% of the maximum number of 2^{31} entries in the SMT. This interesting phenomenon is known as the *birthday paradox*.**

6.1.2 Usability concerns

There is a concern about how the application employing the Sparse Merkle Tree will handle collisions. From the contracts, we can see that a collision will cause the corresponding leaf operation to revert. It may depend on the dApp in question whether there is a way to circumvent the collision at the application level. With collisions being a virtual certainty for applications that use over 250,000 users, we believe these remarks are worth considering.

6.2 Approach: Study on the number of collisions expected according to the number of identities added to the tree

In this section, we focus on simulating how the number of collisions in the sparse Merkle Tree will behave as we add new random identities. To study the collision problem, we created software to simulate the collisions whose pseudo-code is explained below.

```
1  Input: output_csv_filename;
2  Output: csv_file;
3
4  #define MAX_SMT_DEPTH 32
5  function simulate( output_csv_filename )
6  begin
7      while (true)
8      begin
9          file f = open(output_csv_filename,"a");
10         for expoent in {0,1,2,3, ... ,31}
11         begin
12             number_leaves_to_be_added = 1 << exponent;
13             smt = create_empty_smt( MAX_SMT_DEPTH );
14             collisions[exponent] = add_random_leaves( smt, number_leaves_to_be_added);
15         end for
16         f.append_to_csv_file(collisions);
17         f.close();
18     end while
19 end function
```

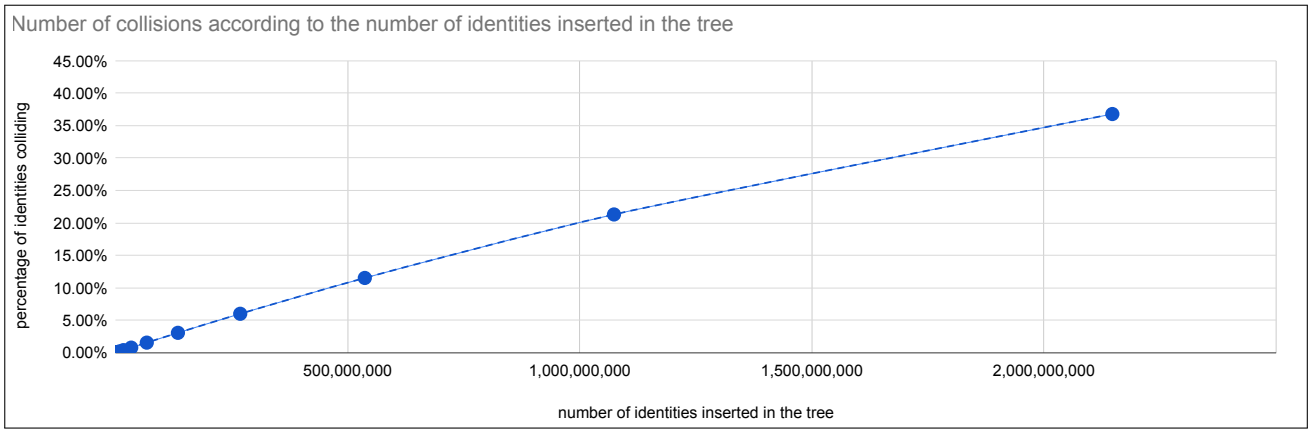


Fig. 5: Collisions expected in the SMT of 32 levels obtained by software simulation

The code receives the `output_csv_filename`. The simulator starts an infinite loop. Line 9 opens the `csv` file for appending data. Line 10 is a `for-loop` over the variable `exponent` that will run from 0 to 31. Line 12 defines the number of leaves to be added to the SMT as 2^{exponent} or simply 1 `<<` exponent. Line 13 creates a new SMT. Line 14 adds `number_leaves_to_be_added` random nodes to the SMT. After each interaction, the function returns the number of collisions, which are then stored in the array `collisions`. After `exponent` looping over all the range from 0 up to 31, line 16 appends the collision data to the `csv` file, and the file is closed to ensure that all data has been saved.

Our team has executed 138 interactions for this report, and the most important results are presented now. Fig. 5 shows the probability of collisions as we add identities to the SMT. Even having 2^{31} leaves (2.147 billion leaves), **when we try to add 500 million leaves**, more than 10% of the requests fail due to a hash clash. At this point, the SMT has less than 450 million identities and almost 1.7 billion empty leaves. **When we try to add 1 billion identities**, more than 20% of the requests fail due to a hash clash. Thus, the SMT ends with less than 800 million identities stored and more than 1.3 billion empty leaves. Finally, **when we try to add 2^{31} leaves**, almost 37% of the requests collide.

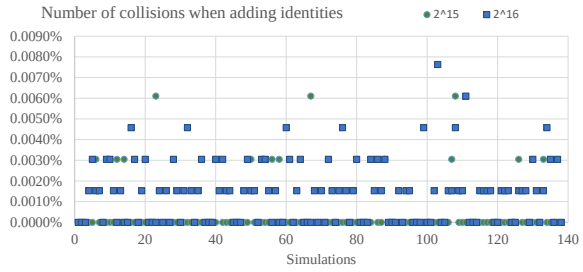
	2^0 - 2^11	2^12	2^13	2^14	2^15	2^16	2^17	2^18	2^19	2^20	2^21
Min Value	0.0000000%	0.0000000%	0.0000000%	0.0000000%	0.0000000%	0.0000000%	0.0000000%	0.0030518%	0.0089645%	0.0199318%	0.0448227%
Max Value	0.0000000%	0.0244141%	0.0000000%	0.0061035%	0.0061035%	0.0076294%	0.0083923%	0.0095367%	0.0156403%	0.0294685%	0.0518322%
Range	0.0000000%	0.0244141%	0.0000000%	0.0061035%	0.0061035%	0.0076294%	0.0083923%	0.0064850%	0.0066757%	0.0095367%	0.0070095%
Standard Deviation	0.0000000%	0.0029177%	0.0000000%	0.0016531%	0.0012088%	0.0014888%	0.0013844%	0.0013716%	0.0013646%	0.0015728%	0.0014185%
Confidence Interval (95%)	0.0000000%	0.0004868%	0.0000000%	0.0002758%	0.0002017%	0.0002484%	0.0002310%	0.0002288%	0.0002277%	0.0002624%	0.0002367%

	2^22	2^23	2^24	2^25	2^26	2^27	2^28	2^29	2^30	2^31
Min Value	0.0941515%	0.1919389%	0.3855288%	0.7731050%	1.5419349%	3.0573435%	5.9935372%	11.5169361%	21.3030718%	36.7861910%
Max Value	0.1018763%	0.1992464%	0.3932714%	0.7800370%	1.5500665%	3.0636482%	6.0016654%	11.5230497%	21.3085684%	36.7893524%
Range	0.0077248%	0.0073075%	0.0077426%	0.0069320%	0.0081316%	0.0063047%	0.0081282%	0.0061136%	0.0054966%	0.0031614%
Standard Deviation	0.0015326%	0.0014611%	0.0015172%	0.0013932%	0.0015073%	0.0012331%	0.0013214%	0.0011778%	0.0010324%	0.0006518%
Confidence Interval (95%)	0.0002557%	0.0002438%	0.0002531%	0.0002324%	0.0002515%	0.0002057%	0.0002205%	0.0001965%	0.0001723%	0.0001088%

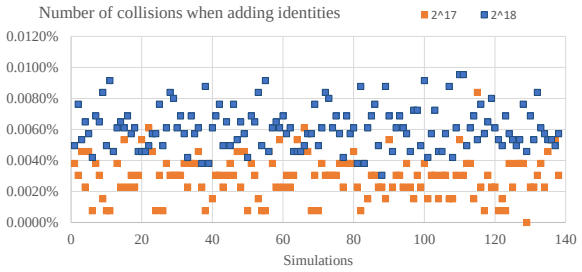
Fig. 6: Table summarizing the collisions according to the number of requests sent to the sparse Merkle Tree with depth=32

Fig. 6 characterizes the results in terms of `minimum value`, `maximum value`, `range`, `standard deviation`, and `confidence interval of 95%`. The `standard deviation` is the degree of dispersion or the scatter of the data points relative to its mean. It tells how the values are spread across the data sample, and it measures the variation of the data points from the mean. The `confidence interval` is the range of values that you expect your estimate to fall between a certain percentage of the time if you run your experiment again or re-sample the population similarly. The confidence level is the percentage of times you expect to reproduce an estimate between the upper and lower bounds of the confidence interval and is set by the α value (in our case, 95%). This figure summarizes the probability of collisions according to the number of requests for adding new identities (leaves). We notice that we have not detected collisions when adding from $2^0(1)$ up to 2^{11} (2.048) identities. Thus, this seems to be a safe interval, although our number of runs was not expressive (only 138 runs). So, **in order to confirm this behavior, we ran another 1 million simulations** considering only the interval $2^0(1)$ up to 2^{11} (2.048), which corresponds to 12 million data points. Considering those 12 million data points, we notice only 4,487 collisions.

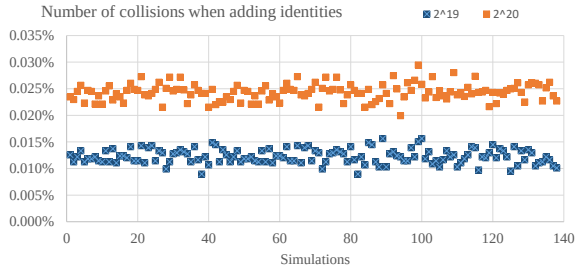
Figs. 7(a) to 7(n) shows the probability of collisions for different numbers of identities added to the sparse Merkle Tree. As we increase the number of identities added, we also increase the probability of collisions.



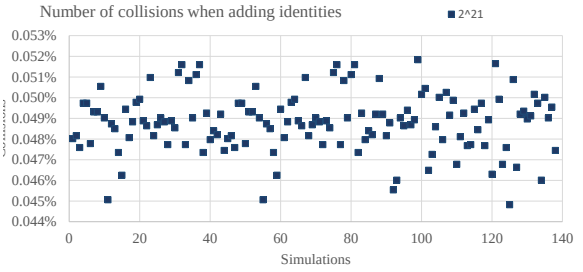
(a) adding 2^{15} and 2^{16} identities



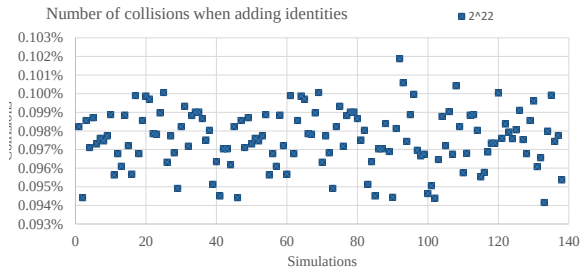
(b) adding 2^{17} and 2^{18} identities



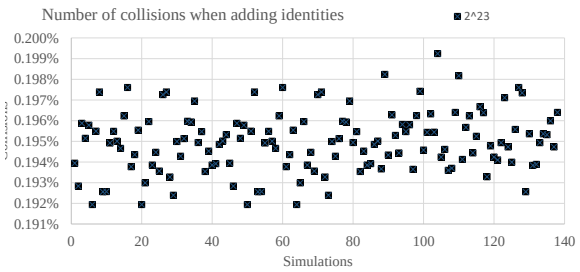
(c) adding 2^{19} and 2^{20} identities



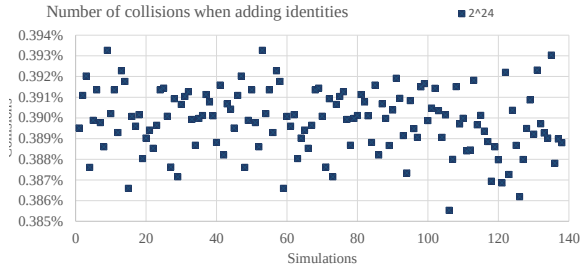
(d) adding 2^{21} identities



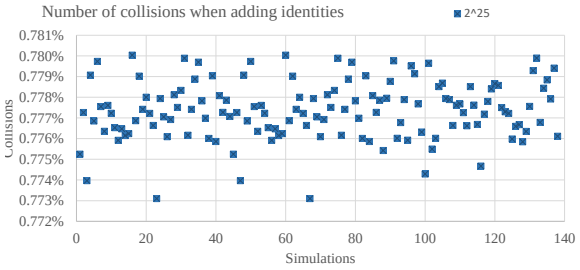
(e) adding 2^{22} identities



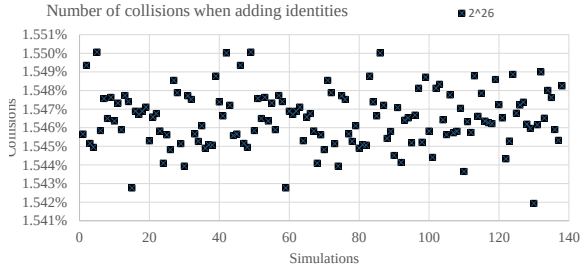
(f) adding 2^{24} identities



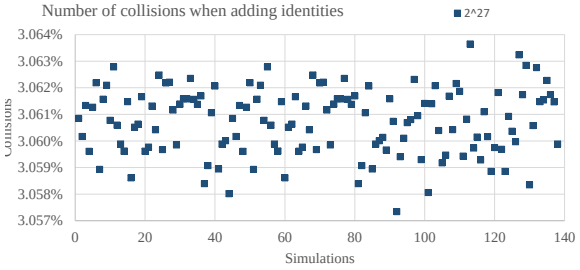
(g) adding 2^{24} identities



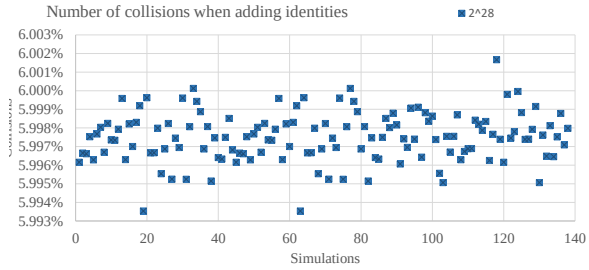
(h) adding 2^{25} identities



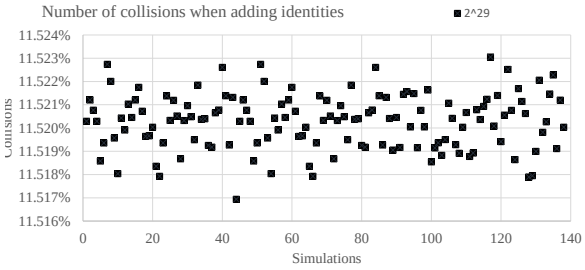
(i) adding 2^{26} identities



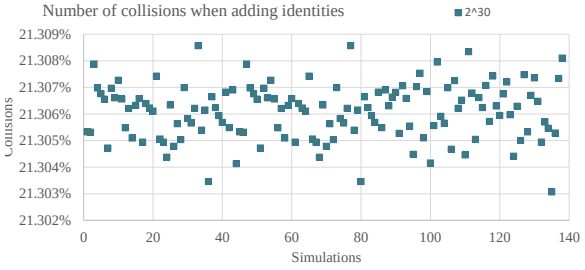
(j) adding 2^{27} identities



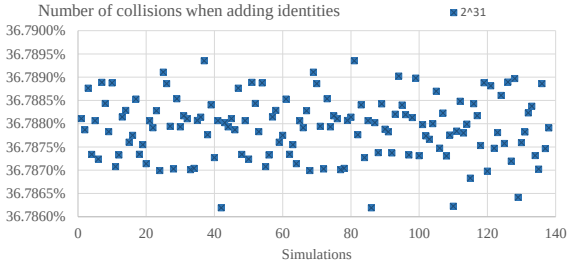
(k) adding 2^{28} identities



(l) adding 2^{29} identities



(m) adding 2^{30} identities



(n) adding 2^{31} identities

Fig. 7: Probability of collisions according to the number of identities inserted in the sparse Merkle Tree.

6.3 Choosing an appropriate depth limit

The graphs and experiments above clearly show the risks behind setting a depth limit of 32 for the SMT. They also show how quickly we may spot collisions in this scenario. In light of this, what would be an appropriate depth limit to balance security and performance? Let us return to Eq. (2). Assume a protocol to handle $n = 2^{20}$ (i.e, roughly 1 million) identities. What is the probability of collision as a function of the depth limit? The graph below illustrates this.

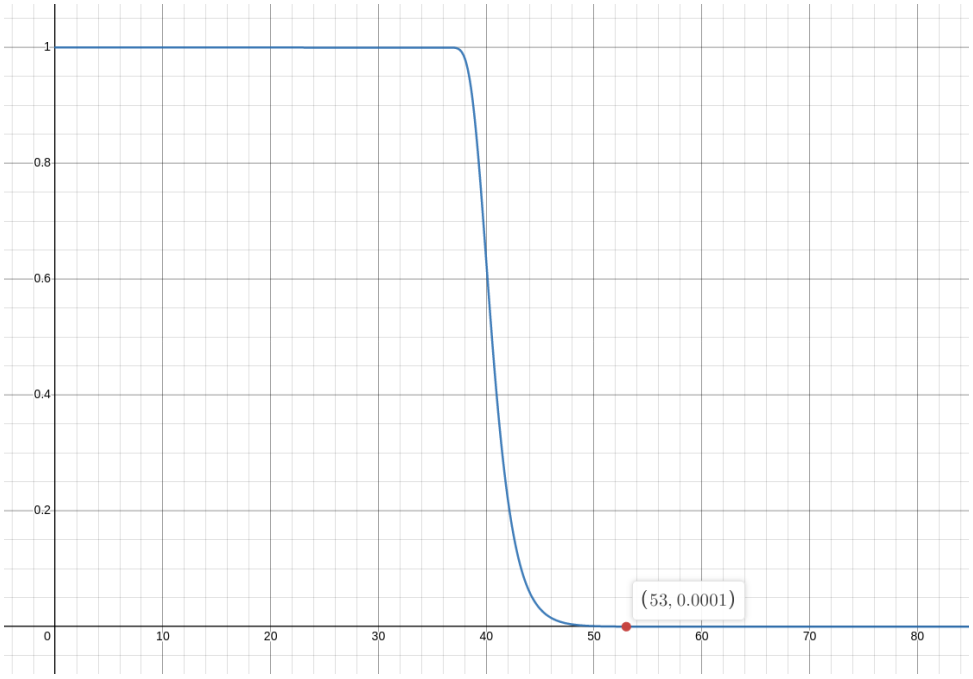


Fig. 8: Estimate the probability of collision P_{col} as a function of the depth limit, assuming a total of $n = 2^{20}$ leaves pushed to the tree. For a depth limit of 53, the highlighted point shows that the collision probability goes down to 0.0001 (0.01%). For a depth limit of 64, the probability goes down even further to 0.000006%.

Fig. 8 suggests depth limits above 50 achieve much greater security regarding collision—if the SMT is expected to hold a million leaves. There are many more permutations of this analysis, all of which can be treated with equation (2). For reference, we show an example below.

Example: Suppose the protocol has grown to $n = 220$ million identities (roughly the total number of Ethereum unique addresses as of January 2023). Compute the probability of collision for a depth limit of 64.

With the aforementioned depth limit, we have $T = 2^{63}$ different leaves. Substituting in (2), we get

$$P_{\text{col}} \approx 1 - e^{-220\times10^6\times(220\times10^6-1)/2\cdot2^{63}} \approx 0.0026 = 0.26\%.$$

We notice that the depth limit of 64 still provides reasonable security for this greatly enhanced identity load.

7 Formal Specification of Sparse Merkle trees

In this section, we introduce a formal specification of a sparse Merkle tree and its operations.

7.1 Preliminaries

First, we introduce some basic sets that we will use throughout the exposition of the specification:

- $\mathbb{N}$, the set of natural numbers.
- `UInt256`, the set of unsigned, 256-bit, integers.

We can now define a basic binary tree data structure Node with two kinds of leaf nodes. We will use this data structure to abstractly model the state of a sparse Merkle tree.

$$\begin{aligned}\texttt{Node} := &\ Empty \\ &\mid Leaf(i : \mathbb{N}, v : \texttt{UInt256}) \\ &\mid Middle(l : \texttt{Node}, r : \texttt{Node})\end{aligned}$$

Note here that child nodes are directly 'stored' within the parent nodes, unlike in Merkle Tree implementations where parent nodes contain as members only hashes of their children. This is done both for clarity and ease of reasoning and to better model the intended structure, which Merkle Trees are but one way of implementing.

Next, we introduce several functions and predicates to assist in reasoning about these trees.

$$\begin{aligned}&\mathbf{nodes} : \texttt{Node} \to \mathcal{P}(\texttt{Node}) \\ &\mathbf{nodes}(Empty) = \{Empty\} \\ &\mathbf{nodes}(Leaf(i, v)) = \{Leaf(i, v)\} \\ &\mathbf{nodes}(Middle(l, r)) = \{Middle(l, r)\} \cup \mathbf{nodes}(l) \cup \mathbf{nodes}(r)\end{aligned}$$

$$\begin{aligned}&\mathbf{leaves} : \texttt{Node} -> \mathcal{P}(\texttt{Node}) \\ &\mathbf{leaves}(n) = \{Leaf(i, v) | Leaf(i, v) \in \mathbf{nodes}(n)\}\end{aligned}$$

$\mathbf{nodes}$ returns a set containing every node reachable from the one given, including itself. Importantly reachability is defined here in the directed sense where children are reachable by parents, but not vice versa. $\mathbf{leaves}$ is then a filter on this that only returns reachable Leaf nodes.

$$\begin{aligned}&\mathbf{distance} : \texttt{Node} \times \texttt{Node} \rightharpoonup \mathbb{N} \\ &\mathbf{distance}(Empty(), Empty()) = 0 \\ &\mathbf{distance}(Leaf(i, v), Leaf(i, v)) = 0 \\ &\mathbf{distance}(Middle(l, r), n) \\ &\quad \mid n \in \mathbf{nodes}(l) = 1 + \mathbf{distance}(l, n) \\ &\quad \mid n \in \mathbf{nodes}(r) = 1 + \mathbf{distance}(r, n) \\ &\quad \mid \texttt{otherwise} = \texttt{undefined}\end{aligned}$$

$\mathbf{distance}$ is a partial function defined if-and-only-if the second node is reachable from the first. It counts the number of edges that must be traversed to get from one to the other.

$$\begin{aligned}&\mathbf{depth} : \texttt{Node} \to \mathbb{N} \\ &\mathbf{depth}(Empty()) = 0 \\ &\mathbf{depth}(Leaf(\_, \_)) = 0 \\ &\mathbf{depth}(Middle(left, right)) = 1 + \mathbf{max}(\mathbf{depth}(left), \mathbf{depth}(right))\end{aligned}$$

The **depth** function provides the distance between a root and its most distant leaf or empty child. A single node is defined as having a depth of 0, and a such limit on depth also tells us how many bits of information from a node's index can be utilized. It is important to note that while this lines up with the use of the depth parameter in recursive functions, it is slightly misaligned with `MAX_SMT_DEPTH`. Due to the condition in `_pushLeaf`, the maximum depth allowed by this definition of a tree is `MAX_SMT_DEPTH` $- 1$. The data structure described by `Node` is, by itself, far too general; no link has yet been made between indices and location in the tree. To fill this gap, we introduce the predicate $\mathbf{pathMatchesIndex} \subseteq Node \times Node \times \mathbb{N}$ which has the following definition:

$$\begin{aligned}
\mathbf{pathMatchesIndex}(n, n, _) &\iff \mathbf{True} \\
\mathbf{pathMatchesIndex}(Middle(l, r), n, 2x+1) &\iff \mathbf{pathMatchesIndex}(r, n, x) \\
\mathbf{pathMatchesIndex}(Middle(l, r), n, 2x) &\iff \mathbf{pathMatchesIndex}(l, n, x)
\end{aligned}$$

Given a tree, a Leaf or Empty node, and an index, this predicate models whether the node is in the correct place given the associated index. We can say that a Node represents a sparse Merkle tree if and only if every leaf, l, reachable from the root satisfies $\mathbf{pathMatchesIndex}(root, l, l.index)$. Importantly no minimum suffix length of the index is defined here, only that by following it, you will reach the intended node. However, this only works if you start at the root of the tree. Many functions used here recur through the tree, and as such, we need to be able to reason about placement with regard to any arbitrary higher node. For this purpose, we define the $\mathbf{sparseMerkleSubtree} \subseteq \texttt{Node}\,\mathbb{N}$ predicates parameterized by the depth at which it is attached to the full tree:

$$\begin{aligned}
\mathbf{sparseMerkleSubtree}(Empty, _) &\iff \mathbf{True} \\
\mathbf{sparseMerkleSubtree}(Leaf(_, _), _) &\iff \mathbf{True} \\
\mathbf{sparseMerkleSubtree}(Middle(l, r), attachmentDepth) &\iff \begin{array}{l}
\mathbf{sparseMerkleSubtree}(left, attachmentDepth + 1) \wedge \\
\mathbf{sparseMerkleSubtree}(right, attachmentDepth + 1) \wedge \\
(\forall l \in \mathbf{leaves}(left).\ l.index\ \&\ 2^{attachmentDepth} = 0) \wedge \\
(\forall l \in \mathbf{leaves}(right).\ l.index\ \&\ 2^{attachmentDepth} \neq 0)
\end{array}
\end{aligned}$$

An attachment depth of 0 means we are treating the given node as the root of the whole tree. Intuitively, the attachment depth determines how long of a suffix of the index to ignore, meaning that we can now reason about proper leaf positioning with regard to any intermediate parent node. In order to properly model the structures constructed by this code we need to assert that there are no extraneous nodes added to the tree. We do this using the $\mathbf{minimumSparseMerkleSubtree} \subseteq \texttt{Node} \times \mathbb{N}$ predicate:

$$\begin{aligned}
\mathbf{minimumSparseMerkleSubtree}(node, attachmentDepth) \iff\ & \mathbf{sparseMerkleSubtree}(node, attachmentDepth) \wedge \\
& \forall Middle(left, right) \in \mathbf{nodes}(node).\ (left = Empty() \implies r \neq Empty) \wedge \\
& \forall x \in \texttt{Node}.\ \mathbf{sparseMerkleSubtree}(x, attachmentDepth) \wedge \\
& \mathbf{leaves}(x) = \mathbf{leaves}(node) \implies \\
& \forall l \in \mathbf{leaves}(node).\ \mathbf{distance}(node, l) \leq \mathbf{distance}(x, l)
\end{aligned}$$

Here assert that the tree contains only those Middle nodes required to distinguish its current leaves from each other. With this in place, we can finally define a predicate, $\mathbf{minimumSparseMerkleTree} \subseteq \texttt{Node}$, that asserts that a given binary tree describes a valid tree:

$$\mathbf{minimumSparseMerkleTree}(x) \iff \mathbf{minimumSparseMerkleSubtree}(x, 0)$$

Finally, to reason about the proofs of inclusion and non-inclusion produced by the various `getProof...` functions, we define our hash function $\mathbf{H}$ and two additional helpers: $\mathbf{H_l}$ and $\mathbf{H_m}$ are two separate hash functions of type `UInt256` $\times$ `UInt256` $\rightarrow$ `UInt256`, representing `PoseidonUint3L.poseidon` with the constant 1 for the third parameter, and `PoseidonUint2L.poseidon` respectively.

$$\begin{aligned}
&\mathbf{H} : \texttt{Node} \rightarrow \texttt{UInt256} \\
&\mathbf{H}(Empty) = 0 \\
&\mathbf{H}(Leaf(i, v)) = \mathbf{H_l}(i, v) \\
&\mathbf{H}(Middle(l, r)) = \mathbf{H_m}(\mathbf{H}(l), \mathbf{H}(r))
\end{aligned}$$

From these two base hash functions, we recursively construct $\mathbf{H}$ to produce hashes for our `Node` type, in line with how the code produces hashes for its `Node` type. The aforementioned helpers are then defined as follows:

$$\begin{aligned}
&\mathbf{hashChain} : \texttt{Node} \times \texttt{Node} \times \mathbb{N} \rightharpoonup [\texttt{UInt256}] \\
&\mathbf{hashChain}(l, l, _) = [] \\
&\mathbf{hashChain}(Middle(l, r), n, 2x + 1) = H(l) : \mathbf{hashChain}(r, n, x) \\
&\mathbf{hashChain}(Middle(l, r), n, 2x) = H(r) : \mathbf{hashChain}(l, n, x)
\end{aligned}$$

$\mathbf{hashChain}$ is used to construct the Merkle proofs of inclusion or non-inclusion. Given a start node, a target node, and an index, it traverses the tree along the path defined by the index, recording the hashes of the paths not taken. As such by using the index to determine the layout and starting with the hash of the target node, one can arrive at the hash of the root by repeated application of the hash function.

$$\begin{aligned}
&\mathbf{pad} : [\texttt{UInt256}] \times \mathbb{N} \rightharpoonup [\texttt{UInt256}] \\
&\mathbf{pad}(xs, len) = ys \iff length(ys) = len \wedge \\
&\quad \forall i \in \mathbb{N}.\ (i < length(xs) \implies xs[i] = ys[i]) \wedge (length(xs) \leq i \wedge i < len \implies ys[i] = 0)
\end{aligned}$$

Since proofs in this implementation always return a list of length `MAX_SMT_DEPTH`, we also require the partial function $\mathbf{pad}$ to take a list and pad it with zeroes if necessary.

7.2 Introduction to Hoare triples and separation logic

Now that we have introduced the abstraction of the state of the module we will use for our specification, we will give a short introduction to the specification format we use. Our functional specification will be written using *Hoare triples*:

$$\forall x \in X. \vdash \{P\}\ C\ \{Q\}$$

Where P and Q are assertions on the machine state, in this case, the state of the EVM, and C is a solidity command, in our case, since we are writing functional specifications for the module's operations, these will always be individual function calls. Intuitively, this specification means that, for an arbitrary assignment of the *logical variable*, $x \in X$, in any state satisfying the assertion P, the *precondition*, if we execute the command C, any terminating, non-faulting execution terminates in a state satisfying Q, the *postcondition*. In *Hoare logic*, these assertions are usually written in some flavor of first-order logic ranging over appropriate programs and logical variables. For example:

$$\texttt{y} = 2\texttt{x} + 7 \wedge \texttt{l} = 1 \implies \texttt{x} = 3$$

This assertion is satisfied in any state where the relation $\texttt{y} = 2\texttt{x} + 7$ holds between the program variables `x` and `y` and, if the program variable `l` has value 1, then $\texttt{x} = 3$, and consequently, $\texttt{y} = 13$. However, we will use *classical separation logic*, an extension of Hoare logic for reasoning about shared, mutable resources. It introduces the *separation conjunction*, $P * Q$, pronounced P *sep* Q. Intuitively, this asserts the *ownership* of the disjoint resources asserted by P and Q. The separating conjunction has a *unit*, `emp`, for which the expected axioms hold:

$$P \iff P * \mathsf{emp} \iff \mathsf{emp} * P$$

We then also introduce *cell assertions*, $E_1 \mapsto E_2$, which, in our case, asserts that the cell at the address that E_1 evaluates to in storage memory has the value that E_2 evaluates to. To enforce that a cell assertion represents *ownership* of the assertions, the following axiom holds:

$$E_1 \mapsto \_ * E_2 \mapsto \_ \wedge E_1 = E_2 \implies \bot$$

This axiom asserts that when $E_1 = E_2$, the assertion $E_1 \mapsto \_ * E_2 \mapsto \_$ is a contradiction, i.e., implies false, $\bot$, as the resources represented by the two cell assertions are not disjoint, they predicate over the same cell in storage memory. Note that _ represents an arbitrary value. Similarly to logical conjunction, $\wedge$, the separating conjunction is also commutative and associative. This allows us to define an *iterated separated conjunction*:

$$\circledast_{x \in X} P(x)$$

This assertion simply represents the separating conjunction of the resources asserted by $P(x)$ for each $x \in X$. Finally, in contrast to logical conjunction, where conjuncts can be eliminated, i.e. $P \wedge Q \implies P$, this is not the case for separating conjunctions. This prevents cell assertions recording updates to the storage memory to be "forgotten". Otherwise, if cell assertions were conjuncted together, using the consequence rule of Hoare logic:

$$\frac{P \implies P' \qquad \vdash \{P'\}\ C\ \{Q'\} \qquad Q' \implies Q}{\vdash \{P\}\ C\ \{Q\}}$$

we could simply forget updates to storage memory that the command C performed. This rules allows us to infer that if $P \implies P'$, i.e., every state satisfying P satisfies P', and $\vdash \{P'\}\ C\ \{Q'\}$, and finally $Q' \implies Q$, then any state satisfying P, since it satisfies P', must be taken by a terminating, non-faulting execution of C to Q', which in turn must satisfied Q, and therefore, we can infer that $\vdash \{P\}\ C\ \{Q\}$ holds. However, since implication does not allow us to forget separated conjuncts, this is no longer a problem. The separation logic also comes with a host of advantages, allowing local, modular, and abstract reasoning. However, an exposition of the details of these properties is outside the scope of this introduction. However, we have now introduced enough of the basics to be able to explain our separation logic specifications for the Sparse Merkle Tree in the following section.

7.3 Specifications

We start with a simple spec for `_addNode`, where we assert that if a hash is currently associated with an Empty node, it can be overwritten with a given node and that the return value is appropriate.

$$\vdash \{\texttt{self.nodes}[\mathbf{H}(\texttt{node})] \mapsto \mathit{Empty}()\}\ \texttt{ret} = \texttt{\_addNode}(\texttt{self}, \texttt{node})\ \{\texttt{self.nodes}[\mathbf{H}(\texttt{node})] \mapsto \texttt{node} \wedge \texttt{ret} = H(\texttt{node})\}$$

With that done we can move onto a spec for `_pushLeaf`. From context within the codebase, we can add some assertions about the inputs: namely that `newLeaf` and `oldLeaf` are distinct leaves and that `pathNewLeaf` and `pathOldLeaf` are their indices. With these properties assumed we can assert that when `_pushLeaf` is run, the minimum sparse Merkle subtree with these two leaves and no others are created, parameterized by `depth` as its attachment depth. Because we need to satisfy the precondition of `_addNode` in order to be able to add these new nodes we need to assert in the precondition of `_pushLeaf` that all of the hashes to be written to currently associate with the Empty node. We assign no specific value to `MAX_SMT_DEPTH` in these specs, but it can be no greater than 257 due to the 256-bit index length, and no less than 2 owing to how the depth restriction in `_pushLeaf` is formed.

$$\begin{array}{l}
\forall afterTree \in \texttt{Node}. \\
\vdash \\
\left\{ \begin{array}{l}
\texttt{self.nodes}[\mathbf{H}(\texttt{oldLeaf})] \mapsto \texttt{oldLeaf} * \\
\quad \circledast_{n \in \mathbf{nodes}(afterTree) \setminus \texttt{oldLeaf}}\ \texttt{self.nodes}[\mathbf{H}(n)] \mapsto \mathit{Empty}() \wedge \\
\mathbf{minimumSparseMerkleSubtree}(afterTree, \texttt{depth}) \wedge \\
\mathbf{leaves}(afterTree) = \{\texttt{oldLeaf}, \texttt{newLeaf}\} \wedge \\
\texttt{oldLeaf} \neq \texttt{newLeaf} \wedge \\
\mathbf{depth}(afterTree) + \texttt{depth} < \texttt{MAX_SMT_DEPTH} \wedge \\
\exists v, v'. \\
\quad \texttt{oldLeaf} = \mathit{Leaf}(\texttt{pathOldLeaf}, v) \wedge \\
\quad \texttt{newLeaf} = \mathit{Leaf}(\texttt{pathNewLeaf}, v')
\end{array} \right\} \\
\texttt{ret} = \texttt{_pushLeaf}(\texttt{self}, \texttt{newLeaf}, \texttt{oldLeaf}, \texttt{depth}, \texttt{pathNewLeaf}, \texttt{pathOldLeaf}) \\
\left\{ \begin{array}{l}
\circledast_{n \in \mathbf{nodes}(afterTree)}\ \texttt{self.nodes}[\mathbf{H}(n)] \mapsto n \wedge \\
\quad \texttt{ret} = \mathbf{H}(afterTree)
\end{array} \right\}
\end{array}$$

Next, we introduce the specification for the `_addLeaf` function. Where before we went from a single leaf to a minimum sparse Merkle subtree with two leaves, we now go from one minimum sparse Merkle subtree to another attaching at the same depth. The block at the end of the precondition states two cases: either the old tree has a leaf with the same index as the new one and it gets replaced, or it does not and the new leaf can coexist alongside all the existing ones as long as doing so doesn't violate the depth assertions:

$$\begin{array}{l}
\forall beforeTree, afterTree \in \texttt{Node}. \\
\vdash \\
\left\{ \begin{array}{l}
\circledast_{n \in \mathbf{nodes}(beforeTree)}\ \texttt{self.nodes}[\mathbf{H}(n)] \mapsto n * \\
\circledast_{n \in \mathbf{nodes}(afterTree) \setminus \mathbf{nodes}(beforeTree)}\ \texttt{self.nodes}[\mathbf{H}(n)] \mapsto \mathit{Empty} \wedge \\
\mathbf{minimumSparseMerkleSubtree}(beforeTree, \texttt{depth}) \wedge \\
\mathbf{minimumSparseMerkleSubtree}(afterTree, \texttt{depth}) \wedge \\
\mathbf{depth}(beforeTree) < \texttt{MAX_SMT_DEPTH} - \texttt{depth} \wedge \\
\mathbf{depth}(afterTree) < \texttt{MAX_SMT_DEPTH} - \texttt{depth} \wedge \\
(\forall n, m \in \mathbf{nodes}(beforeTree) \cup \mathbf{nodes}(afterTree).\ n \neq m \implies \mathbf{H}(n) \neq \mathbf{H}(m)) \wedge \\
\left(\begin{array}{l}
\left(\begin{array}{l}
\exists l \in \mathbf{leaves}(beforeTree). \\
\quad l.index = \texttt{newLeaf.index} \wedge \mathbf{leaves}(afterTree) = \{\texttt{newLeaf}\} \cup \mathbf{leaves}(beforeTree) \setminus \{l\}
\end{array} \right) \vee \\
\left(\begin{array}{l}
\mathbf{leaves}(afterTree) = \{\texttt{newLeaf}\} \cup \mathbf{leaves}(beforeTree) \wedge \\
\forall l \in \mathbf{leaves}(beforeTree).\ l.index \neq \texttt{newLeaf.index}
\end{array} \right)
\end{array} \right)
\end{array} \right\} \\
\texttt{ret} = \texttt{_addLeaf}(\texttt{self}, \texttt{newLeaf}, \texttt{oldLeaf}, \texttt{nodeHash}, \texttt{depth}) \\
\left\{ \begin{array}{l}
\circledast_{n \in \mathbf{nodes}(afterTree) \cup \mathbf{nodes}(beforeTree)}\ \texttt{self.nodes}[\mathbf{H}(n)] \mapsto n \wedge \\
\texttt{ret} = \mathbf{H}(afterTree)
\end{array} \right\}
\end{array}$$

To finish this sequence, we specify the top-level function: `add`. Much of this will look familiar from `_addLeaf` with the difference that we're now working with minimum sparse Merkle trees, not subtrees. Additionally, we specify the change to the `rootHistory` list and `rootEntries`.

$$
\begin{array}{l}
\forall beforeTree, afterTree \in \texttt{Node}, rootHistoryLength \in \mathbb{N}, beforeTreeRootEntry \in RootEntry. \\
\vdash \\
\left\{ \begin{array}{l}
\texttt{self.rootHistory.length} \mapsto rootHistoryLength * \\
\left(\begin{array}{l} (\texttt{self.rootHistory}[rootHistoryLength - 1] \mapsto \mathbf{H}(beforeTree) \wedge rootHistoryLength > 0) \vee \\ (\texttt{emp} \wedge rootHistoryLength = 0) \end{array} \right) * \\
\texttt{self.rootEntries}[\mathbf{H}(beforeTree)] \mapsto beforeTreeRootEntry * \\
\texttt{self.rootEntries}[\mathbf{H}(afterTree)] \mapsto \_ * \\
\circledast_{n \in \mathbf{nodes}(beforeTree)} \texttt{self.nodes}[\mathbf{H}(n)] \mapsto n * \\
\circledast_{n \in \mathbf{nodes}(afterTree) \setminus \mathbf{nodes}(beforeTree)} \texttt{self.nodes}[\mathbf{H}(n)] \mapsto Empty \wedge \\
\mathbf{minimumSparseMerkleTree}(beforeTree) \wedge \\
\mathbf{minimumSparseMerkleTree}(afterTree) \wedge \\
\mathbf{depth}(beforeTree) < \texttt{MAX\_SMT\_DEPTH} \wedge \\
\mathbf{depth}(afterTree) < \texttt{MAX\_SMT\_DEPTH} \wedge \\
(\forall n, m \in \mathbf{nodes}(beforeTree) \cup \mathbf{nodes}(afterTree).\ n \neq m \implies \mathbf{H}(n) \neq \mathbf{H}(m)) \wedge \\
\left(\begin{array}{l} \left(\begin{array}{l} \exists l \in \mathbf{leaves}(beforeTree). \\ \quad l.index = \texttt{i} \wedge \\ \quad \mathbf{leaves}(afterTree) = \{Leaf(\texttt{i}, \texttt{v})\} \cup \mathbf{leaves}(beforeTree) \setminus \{l\} \end{array} \right) \vee \\ \left(\begin{array}{l} \mathbf{leaves}(afterTree) = \{Leaf(\texttt{i}, \texttt{v})\} \cup \mathbf{leaves}(beforeTree) \wedge \\ \forall l \in \mathbf{leaves}(beforeTree).\ l.index \neq \texttt{i} \end{array} \right) \end{array} \right)
\end{array} \right\} \\
\texttt{add}(\texttt{self}, \texttt{i}, \texttt{v}) \\
\left\{ \begin{array}{l}
\texttt{self.rootHistory.length} \mapsto rootHistoryLength + 1 * \\
\left(\begin{array}{l} (\texttt{self.rootHistory}[rootHistoryLength - 1] \mapsto \mathbf{H}(beforeTree) \wedge rootHistoryLength > 0) \vee \\ (\texttt{emp} \wedge rootHistoryLength = 0) \end{array} \right) * \\
\texttt{self.rootHistory}[rootHistoryLength] \mapsto \mathbf{H}(afterTree) * \\
\left(\begin{array}{l} (\texttt{self.rootEntries}[\mathbf{H}(beforeTree)] \mapsto beforeTreeRootEntry \wedge beforeTree = Empty) \vee \\ (\texttt{self.rootEntries}[\mathbf{H}(beforeTree)] \mapsto beforeTreeRootEntry replacedByRoot = \mathbf{H}(afterTree) \wedge beforeTree \neq Empty) \end{array} \right) * \\
\texttt{self.rootEntries}[\mathbf{H}(afterTree)] \mapsto RootEntry \left(\begin{array}{l} replacedByRoot = 0, \\ createdAtTimestamp = \#block\_timestamp, \\ createdAtBlock = \#block\_number \end{array} \right) * \\
\circledast_{n \in \mathbf{nodes}(afterTree) \cup \mathbf{nodes}(beforeTree)} \texttt{self.nodes}[\mathbf{H}(n)] \mapsto n
\end{array} \right\}
\end{array}
$$

Finally, we can move on to the proofs of inclusion and exclusion. Much of the actual work here is done in our definition of the **hashChain** function, so all that is left is to reason about relevant storage variables and the cases for the return value. We outline each in turn, separated by whether looking for the given index finds the Leaf in question, an Empty, or a different Leaf. With this done the rest of the `getProof` functions become relatively easy to specify by using this one as a template and bolting on additional logic as to which root is used.

$$
\begin{array}{l}
\forall tree \in \texttt{Node}, rootIndex \in \mathbb{N}. \\
\vdash \\
\left\{ \begin{array}{l}
\texttt{self.rootHistory}[rootIndex] \mapsto historicalRoot * \\
\quad \circledast_{n \in \mathbf{nodes}(tree)} \quad \texttt{self.nodes}[\mathbf{H}(n)] \mapsto n \wedge \\
\texttt{historicalRoot} = \mathbf{H}(tree) \wedge \\
\mathbf{depth}(tree) < \texttt{MAX\_SMT\_DEPTH}
\end{array} \right\} \\
\texttt{proof} = \texttt{getProofByRoot}(\texttt{self}, \texttt{index}, \texttt{historicalRoot}) \\
\left\{ \begin{array}{l}
\texttt{self.rootHistory}[rootIndex] \mapsto \texttt{historicalRoot} * \\
\quad \circledast_{n \in \mathbf{nodes}(tree)} \quad \texttt{self.nodes}[\mathbf{H}(n)] \mapsto n \wedge \\
\texttt{proof.root} = \texttt{historicalRoot} \wedge \\
\texttt{proof.index} = \texttt{index} \wedge \\
\left(\begin{array}{l}
\exists n \in \mathbf{leaves}(tree). \\
\quad n.index = \texttt{index} \wedge \texttt{proof.existence} = 1 \wedge \\
\quad \texttt{proof.siblings} = \mathbf{pad}(\mathbf{hashChain}(tree, n, \texttt{index}), \texttt{MAX\_SMT\_DEPTH}) \wedge \\
\quad \texttt{proof.value} = n.value \wedge \\
\quad \texttt{proof.auxExistence} = 0 \wedge \\
\quad \texttt{proof.auxIndex} = 0 \wedge \\
\quad \texttt{proof.auxValue} = 0
\end{array} \right) \vee \\
\left(\begin{array}{l}
(\forall n \in \mathbf{leaves}(tree).\ n.index \neq \texttt{index} \wedge \neg \mathbf{pathMatchesIndex}(tree, n, \texttt{index})) \wedge \\
\texttt{proof.existence} = 0 \wedge \\
\texttt{proof.siblings} = \mathbf{pad}(\mathbf{hashChain}(tree, Empty(), \texttt{index}), \texttt{MAX\_SMT\_DEPTH}) \wedge \\
\texttt{proof.auxExistence} = 0
\end{array} \right) \vee \\
\left(\begin{array}{l}
(\forall m \in \mathbf{leaves}(tree).\ m.index \neq index) \wedge \\
\exists n \in \mathbf{leaves}(tree). \\
\quad \mathbf{pathMatchesIndex}(tree, n, \texttt{index}) \wedge \\
\quad \texttt{proof.existence} = 0 \wedge \\
\quad \texttt{proof.siblings} = \mathbf{pad}(\mathbf{hashChain}(tree, n, \texttt{index}), \texttt{MAX\_SMT\_DEPTH}) \wedge \\
\quad \texttt{proof.auxExistence} = 1 \wedge \\
\quad \texttt{proof.auxIndex} = n.index \wedge \\
\quad \texttt{proof.auxValue} = n.value
\end{array} \right)
\end{array} \right\}
\end{array}
$$

8 Risk Rating Methodology

The risk rating methodology used by Nethermind follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood is a measure of how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding other factors are also considered. These can include but are not limited to: Motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if the finding were to be exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue can cause significant damage such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to formally pass to the client;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

9 Issues & Points of Attention: Audit 1

9.1 [Low] Lack of a two-step process for transferring ownership

File(s): `contracts/StateV2.sol`

Description: The `StateV2` contract inherits from OpenZeppelin's `OwnableUpgradeable` contract, which provides a one-step ownership transfer. Transferring ownership in a single step is error-prone and can severely harm the protocol if a mistake happens.

Recommendation(s): We suggest implementing a two-step process for transferring ownership, such as the propose-accept scheme. Check the `Ownable2Step.sol` contract from OpenZeppelin.

Status: Fixed.

Update from the client: We've inherited the `StateV2` contract from `Ownable2StepUpgradeable` instead of `OwnableUpgradeable`. However, it discovered a design flaw, which may cause issues in future upgrades. The reason is that `StateV2` may change or use some new parent contracts in the future, which introduce their state variables. In that way, it may shift down the storage layout. To mitigate that, we've introduced a `uint256[500] __gap;` array as the first state variable in the `StateV2` contract, which can be reduced or extended in length to mitigate such cases. Please let us know if there is a better or standard solution for such a case.

Update from Nethermind: Fixed in commit hash `d9be60d7c92d331058135f4fa124969fb02fdcf3`. The private variable `__gap` is a good solution for ensuring the security of the storage variables' layout in case of future updates.

9.2 [Low] Unnecessary space allocation in `Proof.siblings`

File(s): `contracts/lib/Smt.sol`

Description: The member `siblings` in the structure `Proof` allocates `MAX_SMT_DEPTH` slots. However, since the maximum number of siblings is `MAX_SMT_DEPTH-1`, the allocated space in `Proof.siblings` can be `MAX_SMT_DEPTH-1`. The current struct definition can be seen in the code snippet below.

```
    /**
     * @dev Struct of the node proof in the SMT
     */
    struct Proof {
        uint256 root;
        bool existence;
        ////////////////////////////////////////////////////////
        // @audit A proof will have MAX_SMT_DEPTH - 1 maximum. //
        ////////////////////////////////////////////////////////
        uint256[MAX_SMT_DEPTH] siblings;
        uint256 index;
        uint256 value;
        bool auxExistence;
        uint256 auxIndex;
        uint256 auxValue;
    }

```

Below we present a test case written in Foundry that proves the issue.

```solidity
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;
import "forge-std/Test.sol";
import "forge-std/console.sol";
import "../src/Smt.sol";
////////////////////////////////////////////////////////////////////
// TESTING ISSUE: "Unnecessary space allocation in Proof.siblings" //
////////////////////////////////////////////////////////////////////
contract SmtTest is Test {
    Smt.SmtData internal smtData;
    using Smt for Smt.SmtData;

    function add(uint256 i, uint256 v) public {
        smtData.add(i, v);
    }

    function getProof(uint256 id) public view returns (Smt.Proof memory) {
        return Smt.getProof(smtData, id);
    }

    function testMAX_DEPTHProofSiblings() public {
        // We add the leaves to the last level 31
        //           |-----------<31 bits>---------|
        // in binary 1111111111111111111111111111111 (1 and 30 ones)
        uint256 index = 2147483647;
        // in binary 0111111111111111111111111111111 (0 and 30 ones)
        uint256 index2 = 1073741823;
        add(index, 1);
        add(index2, 2);
        Smt.Proof memory proof = getProof(index);
        // this is the last possible sibling and is non-zero (the hash of leaf at index2)
        assert(proof.siblings[30] == Smt.getNodeHash(Smt.Node(Smt.NodeType.LEAF, 0, 0, index2, 2)));
        // the last slot is empty, no "Out of bounds array access", which means that this space was allocated but never
 ↪  used
        assert(proof.siblings[31] == 0);
    }
    function testFailMAX_DEPTH() public {
        // JUST TO DEMONSTRATE THAT ERROR "Max depth reached" OCCURS CORRECTLY FOR LEVEL 32
        //           |-----------<32 bits>----------|
        // in binary 11111111111111111111111111111111 (1 and 31 ones)
        uint256 index = 4294967295;
        // in binary 01111111111111111111111111111111 (0 and 31 ones)
        uint256 index2 = 2147483647;
        add(index, 1);
        add(index2, 2);
    }
}
```

Recommendation(s): Review the number of siblings required.

Status: Fixed.

Update from the client: The logic of the `MAX_SMT_DEPTH` constant was changed to coincide with the number of siblings. E.g., if the number `MAX_SMT_DEPTH = 32`, then the number of siblings is `32` too. Note, the root tree level has zero depth number, the lowest possible level equal to `MAX_SMT_DEPTH`.

Update from Nethermind: After the client update on the `MAX_SMT_DEPTH` logic, this issue is `Fixed` since the maximum depth of the tree is `MAX_SMT_DEPTH` and not `MAX_SMT_DEPTH - 1`. Therefore the space allocated in `Proof.siblings` is necessary. The update includes changes in the function `_pushLeaf(...)` and the loop in the function `getProofByRoot(...)`. The changes are shown below.

```
function _pushLeaf(...) internal returns (uint256) {
-    if (depth > MAX_SMT_DEPTH - 2) {
-       revert("Max depth reached");
-   }
+    if (depth >= MAX_SMT_DEPTH) {
+       revert("Max depth reached");
+   }
    ...
}
```

```
function getProofByRoot(...)
    public
    view
    onlyExistingRoot(self, historicalRoot)
    returns (Proof memory)
{
    ...
-   for (uint256 i = 0; i < MAX_SMT_DEPTH; i++) {...}
+   for (uint256 i = 0; i <= MAX_SMT_DEPTH; i++) {...}
    ...
```

Both changes allow adding `2^MAX_SMT_DEPTH` leaves to the tree and proof creation from `MAX_SMT_DEPTH` nodes.

9.3 [Info] Code and specification not matching

File(s): contracts/StateV2.sol

Description: The comments for functions `getGISTProofByTime(...)`, `getGISTProofByBlock(...)`, `getGISTRootInfoByBlock(...)`, and `getGISTRootInfoByTime(...)` state that the fetched proof "existed at some block/timestamp or later", which indicates that the proof existed for a root in block/timestamp equal or greater than the one provided. We reproduce one of these comments below.

```
/**
 * @dev Retrieve GIST inclusion or non-inclusion proof for a given identity
 * for GIST root existed in some block or later.
 * @param id Identity
 * @param blockNumber Blockchain block number
 * @return The GIST inclusion or non-inclusion proof for the identity
 */
function getGISTProofByBlock(uint256 id, uint256 blockNumber)
```

However, these functions use the `binarySearchUint256(...)` to select which root will be used. The function `binarySearchUint256(...)` will return a block/timestamp not greater than the one provided. For instance, if we invoke the function `binarySearchUint256(...)` with the input parameter `2000`, and there are roots for the blocks/timestamps `[1000, 1500, 1800, 2010]`, the function returns the information related to the root at block/timestamp `1800`. Below we present a fuzzy test written in Foundry, checking that the function `binarySearchUint256(...)` returns roots created at a timestamp equal to or lower than the one provided.

```
function test_BinarySearchNeverReturnsAGreaterValue(uint256[] memory values, uint256 value) public {
    // We don't want to test arrays with no values
    vm.assume(values.length > 0);

    // If all the roots are inserted after the time we are searching, the timestamp returned will be zero
    vm.assume(value > values[0]);

    // Adding nodes to the tree in different timestamps
    for (uint256 i = 0; i < values.length; i++) {
        // Increase timestamp before adding a root.
        // Modulo operation is used for bounding the value
        skip(values[i] % 10000000000);
        // Add the node
        smt.add(i, i*i);
    }

    // Avoid errNoFutureAllowed error
    value = value % block.timestamp;

    // If all the roots are inserted after the time we are searching, the timestamp returned will be zero
    vm.assume(value > values[0] % 10000000000);

    // Function to test
    Smt.RootInfo memory root = smt.getRootInfoByTime(value);

    // Ensure returned root has a timestamp lower or equal
    assert(root.createdAtTimestamp <= value);
}
```

Recommendation(s): The development team should clarify this use case and update the comments related to the functions `getGISTProofByTime(...)`, `getGISTProofByBlock(...)`, `getGISTRootInfoByBlock(...)`, and `getGISTRootInfoByTime(...)`. If those comments are correct, the team should update the function `binarySearchUint256(...)` to return the values stated in those comments.

Status: Fixed.

Update from the client: The comments were not correct.

Update from Nethermind: Fixed in commit hash 80398a1b46c4108ee9dcccb710486e7a3bc5ec99.

9.4 [Info] Nodes with incorrect depth are allowed

File(s): contracts/lib/Smt.sol

Description: The function `_addLeaf(...)` contains a check for ensuring that leaves with a depth greater than `MAX_SMT_DEPTH` are not added.

```
function _addLeaf(...) internal returns (uint256) {
    if (depth > MAX_SMT_DEPTH) {
        revert("Max depth reached");
    }
    ...
}
```

The maximum amount of levels the tree may have is `MAX_SMT_DEPTH`, which means that the maximum depth of a node is `MAX_SMT_DEPTH - 1`. However, this check potentially allows nodes with a depth of `MAX_SMT_DEPTH` to be added. This finding is rated as `Info` because it is not possible to add leaves at this depth. But it still can create false assumptions. Below we present a test case written in Foundry that proves the issue.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;
import "forge-std/Test.sol";
import "forge-std/console.sol";
import "../src/Smt2.sol";
import "../src/Smt.sol";
////////////////////////////////////////////////////////
// TESTING ISSUE: Nodes with incorrect depth are allowed  //
////////////////////////////////////////////////////////
contract SmtTest2 is Test {
    Smt2.SmtData internal smtData;
    using Smt2 for Smt2.SmtData;

    ////////////////////////////////////////////////////////////////////////////////
    // NOTE: Smt2.sol is changed: _pushLeaf() has removed revert check to test this  //
    ////////////////////////////////////////////////////////////////////////////////

    function add(uint256 i, uint256 v) public {
        smtData.add(i, v);
    }

    function testMAX_DEPTHSmt2() public {
        //           |-----------<32 bits>----------|
        // in binary 11111111111111111111111111111111 (1 and 31 ones)
        uint256 index = 4294967295;
        // in binary 01111111111111111111111111111111 (0 and 31 ones)
        uint256 index2 = 2147483647;

        // Added with _addLeaf(): 1st level is reached
        add(index, 1);
        // Added with _pushLeaf(): 32 level reached, because index and index2 share 31 bits and 32nd is different
        // _pushLeaf() doesn't revert because of the removed check
        add(index2, 2);
        ////////////////////////////////
        // DEMONSTRATION OF THE ISSUE //
        ////////////////////////////////
        // Changed with _addLeaf: this node should not be accessible by _addLeaf(...), but it is
        add(index2, 3);
    }
}
```

Recommendation(s): This issue presents inconsistency. Consider changing the condition to be aligned with the rest of the code.

Status: Fixed.

Update from the client: The fix applied to the previous note also fixes this. We've changed the logic of the `MAX_SMT_DEPTH` constant, and now it is possible to add leaves at the lowest level of the tree.

Update from Nethermind: After the client update on the `MAX_SMT_DEPTH` logic, this issue is `Fixed` since the maximum depth of the tree is `MAX_SMT_DEPTH` and not `MAX_SMT_DEPTH - 1`. Therefore the check in the function `_addLeaf(...)` is correct. The update includes changes in the function `_pushLeaf(...)`, and in the loop of the function `getProofByRoot(...)`.

```
function _pushLeaf(...) internal returns (uint256) {
-    if (depth > MAX_SMT_DEPTH - 2) {
-       revert("Max depth reached");
-   }
+    if (depth >= MAX_SMT_DEPTH) {
+       revert("Max depth reached");
+   }
    ...
}
```

```
function getProofByRoot(...)
    public
    view
    onlyExistingRoot(self, historicalRoot)
    returns (Proof memory)
{
    ...
-   for (uint256 i = 0; i < MAX_SMT_DEPTH; i++) {...}
+   for (uint256 i = 0; i <= MAX_SMT_DEPTH; i++) {...}
    ...
```

Both changes allow adding `2^MAX_SMT_DEPTH` leaves to the tree and proof creation from `MAX_SMT_DEPTH` nodes.

9.5 [Info] Owner can change the verification logic after the contract's deployment

File(s): `contracts/StateV2.sol`

Description: The contract `StateV2` contains an `initialize(...)` function called during deployment time, and it takes the `verifier` address as an input parameter, as reproduced below.

```
function initialize(IVerifier verifierContractAddr) public initializer {
    ////////////////////////////////////////////////////////////
    // @audit Setting the address of the verifier contract    //
    ////////////////////////////////////////////////////////////
    verifier = verifierContractAddr;
    __Ownable_init();
}
```

The `verifier` contract contains the zero-knowledge verification logic. The owner can change this logic after the contract has been deployed. In case the private key of the owner is compromised, an attacker will be able to alter the logic of verification of the smart contract to add illegitimate states in the sparse Merkle Tree. The function `setVerifier(...)` is reproduced below.

```
function setVerifier(address newVerifierAddr) public onlyOwner {
    verifier = IVerifier(newVerifierAddr);
}
```

Recommendation(s): No special action is required. This issue is just highlighting this behavior to users of the protocol.

Status: Acknowledged.

9.6 [Info] Privileged Roles and Ownership

File(s): `contracts/StateV2.sol`

Description: Smart contracts often have `owner` variables to designate the person with special privileges to modify the smart contract.

Recommendation(s): This centralization of power needs to be made clear to the users, especially depending on the level of privilege the contract allows to the owner.

Status: Acknowledged.

9.7 [Info] Upgradability

File(s): contracts/StateV2.sol

Description: Users should be aware that the contract can be upgraded at any given time.

Recommendation(s): Communicate to users the reasons for upgrades beforehand.

Status: Acknowledged

9.8 [Best Practices] Avoidable reversion in function `getRootHistory(...)`

File(s): contracts/lib/Smt.sol

Description: The function `getRootHistory(...)` is a view function responsible for returning the history of all the sparse Merkle tree roots stored in the protocol. It receives the `startIndex` and the `length` to be returned. The `endIndex` is computed by adding `length` to the `startIndex`. When the `endIndex` is greater than the array length, the function reverts. The reversion can be avoided by computing the `endIndex` considering the smallest value between `startIndex + length` and `self.rootHistory.length` as presented below.

```
uint256 endIndex = min(startIndex + length, self.rootHistory.length);
```

By following this approach, this function will never revert due to input values leading to an `Out of the bonds` exception. The code snippet with audit comments is presented below.

```
function getRootHistory(SmtData storage self, uint256 startIndex, uint256 length ) ... returns (RootInfo[] memory)
{
    ...
    ////////////////////////////////////////////////////////////////
    // @audit "endIndex" can be computed as:
    // "min(startIndex + length, self.rootHistory.length)"
    ////////////////////////////////////////////////////////////////
    uint256 endIndex = startIndex + length;
    require(endIndex <= self.rootHistory.length, "Out of bounds of root history");

    RootInfo[] memory result = new RootInfo[](length);
    uint64 j = 0;
    for (uint256 i = startIndex; i < endIndex; i++) {
        uint256 root = self.rootHistory[i];
        result[j] = getRootInfo(self, root);
        j++;
    }
    return result;
}
```

Recommendation(s): Consider computing the `endIndex` so that the function does not have an `Out of the bonds` exception, increasing the user experience. The proposed changes are presented below.

```
- uint256 endIndex = startIndex + length;
- require(endIndex <= self.rootHistory.length, "Out of bounds of root history");
+ uint256 endIndex = min(startIndex + length, self.rootHistory.length);
```

Status: Fixed.

Update from the client: Fixed for both `StateV2.getStateInfoHistoryById()` and `Smt`.getRootHistory().

Update from Nethermind: Fixed in commit hash ebe466957f4d8af8e2c11ad5249e3ac219b9145d.

9.9 [Best Practices] Functions that can have `external` visibility instead of `public`

File(s): `contracts/StateV2.sol`

Description: The functions listed below can have `external` visibility since they are never called inside the contract. Besides setting the proper visibility, `external` functions consume less gas than public ones. The functions are listed below.

```
StateV2.transitState(...)
StateV2.getStateInfoById(...)
StateV2.getStateInfoHistoryLengthById(...)
StateV2.getVerifier()
StateV2.setVerifier()
```

Recommendation(s): Make the functions above `external` instead of `public`.

Status: Fixed.

Update from Nethermind: Fixed in commit hash `c9c661ef00811b8f1eb9367519d4f230aa1b2842`.

9.10 [Best Practices] Memory variables should be initialized

File(s): `contracts/lib/Smt.sol`

Description: The code has some variables that have not been initialized because they are expected to be zero. However, Solidity does not guarantee that memory variables are initialized as zero, as shown here.

Recommendation(s): Always initialize memory variables.

Status: Fixed.

Update from Nethermind: Fixed in commit hash `31926bd130ba6494316a57427ff994816e66fefa`.

9.11 [Best Practices] Not checking the verifier contract for `address(0x0)`

File(s): `contracts/StateV2.sol`

Description: The function `setVerifier(...)` does not check the `newVerifierAddr` for `address(0x0)`. The code is reproduced below.

```
function setVerifier(address newVerifierAddr) public onlyOwner {
    ////////////////////////////////////////////////////////////
    // @audit Not checking "newVerifierAddr" for address(0x0)
    ////////////////////////////////////////////////////////////
    verifier = IVerifier(newVerifierAddr);
}
```

Recommendation(s): Consider checking `newVerifierAddr` for `address(0x0)`.

Status: Acknowledged.

Update from Nethermind: The issue was fixed in commit hash `5b1d9d14db5836fc6d6d1ce4db9ad9c77b2f9e3f` by checking if `newVerifierAddr != 0`. However, the fix was reverted in commit hash `05af5b4414fcd871972992ddc839d62f4ff37a97` because it allows the owner to prevent any new state change.

9.12 [Best Practices] Redundant input arguments in function _pushLeaf(...)

File(s): contracts/lib/Smt.sol

Description: The index of a leaf is also the path to that leaf in the tree. The values for the arguments pathNewLeaf and pathOldLeaf in the function _pushLeaf(...) will share the same values as newLeaf.index and oldLeaf.index. These arguments can be removed from the function signature. The code snippet below shows how the same values are used for those arguments.

```
function _addLeaf(...) internal returns (uint256) {
    ...
    if (node.nodeType == NodeType.EMPTY) {
        leafHash = _addNode(self, newLeaf);
    } else if (node.nodeType == NodeType.LEAF) {
        leafHash = node.index == newLeaf.index
            ? _addNode(self, newLeaf)
    ////////////////////////////////////////////////////////////////////////
    //@audit - The values "newLeaf.index" and "node.index" were already accessible
    //         from the "newLeaf" and the node arguments
    ////////////////////////////////////////////////////////////////////////

            : _pushLeaf(
                self,
                newLeaf,
                node,
                depth,
                newLeaf.index,
                node.index
            );
    } ...

```

Recommendation(s): Remove redundant arguments from the function _pushLeaf(...).

Status: Fixed.

Update from Nethermind: Fixed in commit hash 347dd01048110f8da49069835a93a6fe0bd389c0.

9.13 [Best Practices] Special values able to be used as normal input

File(s): contracts/StateV2.sol

Description: The StateV2 contract considers some values *special* and contract behavior can change if these special values are encountered. In the function transitState(...) an user cannot submit a new identity state when it matches any other identity state that has already been submitted. The check is reproduced below.

```
require(!stateExists(newState), "New state should not exist")
```

The function stateExists(...) checks whether a state exists by using the StateEntry.id value. If the id is not equal to zero, then the state exists. This means an id of zero is treated as an exceptional value. For all practical purposes, it is impossible to have an identity id of zero. However, the function transitState(...) does not prevent this. This issue also applies to the argument newState. When a state is replaced, the struct member replacedBy will contain the newState, which could be zero. This could lead to the StateEntry incorrectly indicating that the state has not been replaced. Again, for all practical purposes, it is impossible to have a newState of zero, but the function transitState(...) does not prevent this.

Recommendation(s): It is considered a best practice to prevent normal inputs from being treated as special values. Therefore consider sanitizing the input and checking if the newState and the id differ from 0.

Status: Fixed.

Update from Nethermind: The issue is partially fixed in commit hash 3245a7563ecc3f8876682b67c965437ce888ca62. The newState is checked for 0 value, but the id is not.

Update from client: Fixed in commit hash ede2e6b3

9.14 [Best Practices] Unnecessary path specification in import

File(s): contracts/lib/Smt.sol

Description: The `Smt.sol` contract imports `Poseidon.sol` as below, which is unnecessary considering that both files are in the same folder.

Recommendation(s): Update the code as shown below.

```
- import "../lib/Poseidon.sol";
+ import "./Poseidon.sol";
```

Status: Fixed.

Update from Nethermind: Fixed in commit hash a04690649188b71cceddc2ef81fcbec54926b7aa.

9.15 [Best Practices] `abicoder v2` pragma is not needed since version `0.8.0`

File(s): contracts/*

Description: The `abicoder v2` is the default version since Solidity `0.8.0`, as shown in the solidity documentation.

Recommendation(s): Remove the `abicoder v2` pragma.

Status: Fixed.

Update from Nethermind: Fixed in commit hash 278e292d1cd7f6063f5ebeb0ef855d00f51af528.

9.16 [Best Practices] Variable can be `uint256` instead of `uint64`

File(s): contracts/lib/Smt.sol

Description: In the function `getRootHistory(...)` the variable `j` is of type `uint64`. Since the EVM word size is 32 bytes long (aligning with `uint256`), working with smaller-sized integer types can incur extra operations when converting the value to the desired format.

```
function getRootHistory(...) public view returns (RootInfo[] memory) {
    ...
    ////////////////////////////////////////////////////////////////
    // @audit "uint64" is more expensive than "uint256"
    ////////////////////////////////////////////////////////////////
    uint64 j = 0;
    for (uint256 i = startIndex; i < endIndex; i++) {
        uint256 root = self.rootHistory[i];
        result[j] = getRootInfo(self, root);
        j++;
    }
    return result;
}
```

Recommendation(s): As there appears to be no reason for `j` to be of type `uint64`, consider changing it to `uint256`.

Status: Fixed.

Update from the Nethermind: Fixed in commit hash 278e292d1cd7f6063f5ebeb0ef855d00f51af528.

10 Issues & Points of Attention: Audit 2

We have conducted a re-audit on the updated implementation of the Polygon team, as they had to refactor the original implementation by changing the data model and design of the `StateV2`. As a result, new issues have been identified, which are listed below. The commit hash of the re-audit is b74788b3e7fb71c2b8c858f4aba0b52aa1d36eae.

10.1 [Info] Inconsistent behavior of getter function for zero root

File(s): contracts/lib/SmtLib.sol

Description: During the initialization of `_gistData`, the first root is assigned a value of `0`. However, when retrieving the proof information, we notice inconsistent behavior between the public functions. Some functions, such as `getProof(...)` and `getProofByRoot(...)`, return the proof for a root value of `0`, while others, such as `getProofByTime(...)` and `getProofByBlock(...)`, revert when the root value is `0`. Below, we introduce the function `getProofByRoot(...)`, which returns a proof when encountering a root of 0:

```
function getProofByRoot(Data storage self, uint256 index, uint256 historicalRoot) public view onlyExistingRoot(self,
↪  historicalRoot) returns (Proof memory) {
    uint256[] memory siblings = new uint256[](self.maxDepth);
    // Solidity does not guarantee that memory vars are zeroed out
    for (uint256 i = 0; i < self.maxDepth; i++) {
        siblings[i] = 0;
    }
    ////////////////////////////////////////////////////////
    // @audit The proof is returned even on root 0
    ////////////////////////////////////////////////////////
    Proof memory proof = Proof({
        root: historicalRoot,
        existence: false,
        siblings: siblings,
        index: index,
        value: 0,
        auxExistence: false,
        auxIndex: 0,
        auxValue: 0
    });

    uint256 nextNodeHash = historicalRoot;
    Node memory node;

    for (uint256 i = 0; i <= self.maxDepth; i++) {
        node = getNode(self, nextNodeHash);
        if (node.nodeType == NodeType.EMPTY) {
            break;
        } else if (node.nodeType == NodeType.LEAF) {
            if (node.index == proof.index) {
                proof.existence = true;
                proof.value = node.value;
                break;
            } else {
                proof.auxExistence = true;
                proof.auxIndex = node.index;
                proof.auxValue = node.value;
                proof.value = node.value;
                break;
            }
        } else if (node.nodeType == NodeType.MIDDLE) {
            if ((proof.index >> i) & 1 == 1) {
                nextNodeHash = node.childRight;
                proof.siblings[i] = node.childLeft;
            } else {
                nextNodeHash = node.childLeft;
                proof.siblings[i] = node.childRight;
            }
        } else {
            revert("Invalid node type");
        }
    }
    return proof;
}
```

Below we present an example of a function that reverts when encountering a root of zero.

```
function getProofByBlock(
    Data storage self,
    uint256 index,
    uint256 blockNumber
) external view returns (Proof memory) {
    RootEntryInfo memory rootInfo = getRootInfoByBlock(self, blockNumber);
    ////////////////////////////////////////////////////////
    // @audit Revert on root 0
    ////////////////////////////////////////////////////////
    require(rootInfo.root != 0, "historical root not found");

    return getProofByRoot(self, index, rootInfo.root);
}
```

Recommendation(s): Ensure consistent behavior among getter functions when providing proof for a root value of `0`.

Status: Fixed

Update from the client: Fixed in commit 6ac5bca7fcd2138f639bcf39722107c0b4e37b9a.

10.2 [Info] NatSpec comment missing in function `calculateBounds(...)`

File(s): contracts/lib/ArrayUtils.sol

Description: In the library `ArrayUtils`, the function `calculateBounds(...)` is lacking NatSpec comments for the input parameter `limit`. Below, we provide the NatSpec comment for the `calculateBounds(...)` function.

```
/**
 * @dev Calculates bounds for the slice of the array.
 * @param arrLength An array length.
 * @param start A start index.
 * @param length A length of the slice.
 ////////////////////////////////////////////////////////
 // @audit Missing description of "limit"
 ////////////////////////////////////////////////////////
 * @return The bounds for the slice of the array.
 */
function calculateBounds(
    uint256 arrLength,
    uint256 start,
    uint256 length,
    uint256 limit
) internal pure returns (uint256, uint256) {
```

Recommendation(s): Please add NatSpec comments for the parameter `limit`.

Status: Fixed

Update from the client: Fixed in commit 6ac5bca7fcd2138f639bcf39722107c0b4e37b9a.

10.3 [Best Pratices] Gaps with round numbers

File(s): contracts/lib/SmtLib.sol, contracts/lib/StateLib.sol

Description: The `__gap` field in the Data struct is used to reserve space for future upgrades. It is considered best practice to allocate a round number of slots in the `__gap` field for ease of memory and reasoning during upgrades. An example of a place for improvement is struct `Data`:

```
struct Data {
    mapping(uint256 => Node) nodes;
    RootEntry[] rootEntries;
    mapping(uint256 => uint256[]) rootIndexes; // root => rootEntryIndex[]
    uint256 maxDepth;
    bool initialized;
    ////////////////////////////////////////////////////////
    // @audit Slots in __gap should be 45 not 48
    ////////////////////////////////////////////////////////
    uint256[48] __gap;
}
```

In this case, it is recommended to allocate `45` slots in the `__gap` field so that the total number of slots used and reserved is a round number. Another example is the struct `Data` in `StateLib`, which should have allocated 48 slots.

```
struct Data {
    mapping(uint256 => Entry[]) stateEntries;
    mapping(uint256 => mapping(uint256 => uint256[])) stateIndexes;
    ////////////////////////////////////////////////////////
    // @audit Slots in __gap should be 48 not 50
    ////////////////////////////////////////////////////////
    uint256[50] __gap;
}
```

Recommendation(s): Consider keeping the space allocated in `__gap` as a round number. Document this practice and the reasoning behind the chosen number of slots in the `__gap` field to ensure clarity and consistency in future upgrades.

Status: Fixed

Update from the client: Fixed in commit 6ac5bca7fcd2138f639bcf39722107c0b4e37b9a.

10.4 [Best Practices] Not testing implementation updates

File(s): `test/state/stateV2.test.ts`

Description: The testing suite does not include a scenario where the implementation of the `StateV2` is changed to a new one.

Recommendation(s): Consider improving the testing suite for `StateV2`.

Status: Acknowledged

10.5 [Best Practices] Uninitialized proxy implementation

File(s): `contracts/state/StateV2.sol`

Description: The implementation contract `StateV2` serves as the logic code for the proxy contract. However, it is possible for any address to directly initialize the implementation contract. To prevent this, it is advisable to include `_disableInitializers(...)` in the constructor, as recommended by OpenZeppelin.

Recommendation(s): Consider including `_disableInitializers(...)` in the constructor.

Status: Fixed

Update from the client: Fixed in commit 6ac5bca7fcd2138f639bcf39722107c0b4e37b9a.

11 Complementary Validations Performed by Nethermind

This section describes each part of the overall audit process. The audit team was divided into groups responsible for the following:

a) Line-by-line inspection of the source code;

b) Manual execution of the algorithm on pen and paper covering all possible code branches;

c) White box testing targeting particular logic;

d) Black box testing.

11.1 White-box tests

White box testing involves testing an application with detailed inside information about its source code, architecture, and configuration. It can expose issues such as security vulnerabilities, broken paths, or data flow issues, which black box testing cannot test comprehensively. **The white-box tests were divided into specific goals:**

a) Validate if every line of code is tested at least once to add a new leaf in the `SMT`;

b) Improve code auditing by scanning the following functions: `add(...)`, `getRoot(...)`, `_addLeaf(...)`, `_pushLeaf(...)`, `_addNode(...)`, and `getNodeHash(...)`.

11.1.1 Methodology

Our methodology consists of two stages:

i) careful inspection of the code, line-by-line, annotating parts of the code that require deeper investigation;

ii) manual code inspection combined with dynamic code analysis during the auditing process. Dynamic analysis is a debugging technique to test and evaluate a program while running, i.e., to apply behavioral analysis on the code for a given input. We extract test case execution traces for monitoring the dynamic interaction. We generate trace files during the execution of the `add(...)` function and their nested functions (`_addLeaf(...)`, `_pushLeaf(...)`, `_addNode(...)`, `getNodeHash(...)`, and `getRoot(...)`). These files present the order of the code that is executed during the test cases, such as called functions, variables, and executed branches for a given input. Then, for each test case, we reinspect the code following the captured traces.

11.1.2 Statement and Branch Coverage

We applied two *White Box Test design techniques*: **i) statement coverage** that is used to verify if every line of code has been executed at least once; and, **ii) branch coverage** that is used to ensure that each decision condition from every branch has been tested at least once.

Branch Coverage: We executed 4 (four) test cases on the instrumented code to validate all branches when calling `smt.add(...)` function and to avoid no branch leading to any unexpected behavior. For example, Fig. 5 describes the existent branches in `_addLeaf` and `_pushLeaf` functions.

```
_addLeaf
  if(depth > MAX_SMT_DEPTH)
      revert()
  if (node.nodeType == NodeType.EMPTY)
      |--- _addNode(...)
  else if (node.nodeType == NodeType.LEAF)
      |--- if(node.index == newLeaf.index)
              |--- _addNode(...)
      |--- else
          |--- _pushLeaf(...)
  else if (node.nodeType == NodeType.MIDDLE)
      |--- if ((newLeaf.index >> depth) & 1 == 1)
              |--- _addLeaf(...)
      |--- else
              |--- _addLeaf(...)
  return leafHash
```

```
_pushLeaf
   if (depth > MAX_SMT_DEPTH - 2)
        revert()
   if ((pathNewLeaf >> depth) & 1 == (pathOldLeaf >> depth)
    ↪  & 1)
        |--- _pushLeaf(...)
        |--- if ((pathNewLeaf >> depth) & 1 == 1)
                newNodeMiddle
        |--- else
                newNodeMiddle
        |--- return _addNode(newNodeMiddle)
   if ((pathNewLeaf >> depth) & 1 == 1)
        |--- newNodeMiddle
   else
        |--- newNodeMiddle
   _addNode(newLeaf);
   return _addNode(newNodeMiddle);
```

Fig. 5: Branches in `_addLeaf` and `_pushLeaf` functions.

Table 2: Test Cases - Branch Coverage obtained by analyzing execution traces.

Index	Binary	Branch Coverage
25763	110010010100011	62%
9379	010010010100011	
3235	110010100011	81%
1187	010010100011	
25763	110010010100011	
9379	010010010100011	

Results: Every branch of the code to add a new leaf node was executed at least once, **except** one condition in `_addLeaf` function related to `MAX_SMT_DEPTH` (see Fig. 5). The above table summarizes some information about the test cases. We defined different indexes to compute statements and the branch coverage measures. Moreover, Table 2 shows the branch coverage for these test cases considering **different values of node**. As we can see, the test case to insert four leaf nodes reached 81% of branch coverage. However, four branches are not tested with these test cases: **i)** When a node already exists with the same index and value; **ii)** When a node already exists but with a different value; **iii)** In `_addLeaf` function when `depth > MAX_SMT_DEPTH` (see Fig. 5); **iv)** In `_pushLeaf` function when `depth > MAX_SMT_DEPTH - 2` (see Fig. 5).

We performed the inputs described in Fig 6 to run the branches described in items (i) and (ii) mentioned above. These tests use the (index, value) pairs to run at least once the branches when adding a new leaf node.

```
leavesToInsert: [
    { i: 9379, v: 100 },
    { i: 9379, v: 100 },
]
```

```
leavesToInsert: [
    { i: 9379, v: 100 },
    { i: 9379, v: 222 },
],
```

Fig. 6: Inputs to test the `add` function when values are the same and different.

Next, we apply tests that fail to test branches when adding a new leaf node process is reverted. **We could not test the branch** when depth is greater than maximum depth in `_addLeaf` function. **This branch is not reachable.** Lastly, to test the branch in `_pushLeaf` function when `depth > MAX_SMT_DEPTH - 2`, we performed the first scenario listed in Table 3 when labeled *"Test that fails"*.

Statement Coverage: We execute the same 4 (four) test cases described in Table 2 and Fig 6 on the instrumented code to ensure that there is no dead code, unused statements, or missing statements. **Result:** All lines of code were executed at least once.

In this work, we combine black-box testing with white-box testing. By combining black-box and white-box testing, testers can achieve a comprehensive "inside-out" inspection of software and increase the coverage of quality aspects and security issues. The black-box tests are discussed in the following subsection.

11.2 Black-box tests

Black-box testing is a method of software testing that examines the functionality of an application without peering into its internal structures or workings. Test cases are built around specifications and requirements, i.e., what the application is supposed to do. Test cases derive from external software descriptions, including specifications, requirements, and design parameters. Although the tests are primarily functional, non-functional tests may also be used. The test designer selects valid and invalid inputs and determines the correct output, often with the help of a test oracle or a previous result known to be correct, without any knowledge of the test object's internal structure.

11.2.1 Boundary Testing

We applied a **Black Box Test design technique** called **Boundary Testing**. This is a type of test in which tests are performed using boundary values. The goal of this test is to evaluate the maximum depth the tree reaches.

The Maximum Depth. The constant `MAX_SMT_DEPTH` set to `n` allows the depth of the tree to reach `n-1` levels, i.e., the insertion of a leaf in the `n-th` level will fail. For this test, we run the `add(...)` function considering `MAX_SMT_DEPTH=32` as defined in the code. Table 3 describes two scenarios: i) Test that fails - Scenario with two indexes where the tree depth reaches 32; and ii) Test that passes - Scenario with two indexes where the tree depth reaches 31.

Table 3: Test Cases for the boundary test

Scenario	Index	Binary
Test that fails	3459916963	11001110001110100010010010100011
	1312433315	01001110001110100010010010100011
Test that passes	1312433315	10011100011101000100100101000011
	238691491	00011100011101000100100101000011

11.3 Final Remarks

Although the provided test suite achieves 96% and 85% coverage on the Branch and Statement measures, there are some caveats that we should take into account. There is no silver bullet to define an absolute number for test coverage. This number depends on the complexity of the requirements, how critical a given implementation is, etc. Typically, achieving 100% coverage is not necessary because usually there are non-critical methods such as `getters`. Instead, it is recommended to carefully evaluate snippets not covered by the tests to ensure they are not relevant or hard to test. During the audit, we observed that **not all branches were tested** by the existent unit tests when **inserting leaf nodes**, even having coverage reaching 96% (Branch metric) and 85% (Statement metric). **The main goal of the tests** applied in this section was to ensure that all the code responsible for adding nodes was executed at least once. Consequently, **avoiding undesirable behavior** during its execution in the production environment. **Result:** During our code analysis, we observe that the branch in the function `_addLeaf(...)` that tests `if depth > MAX_SMT_DEPTH` is not reachable.

12 Documentation Evaluation

Technical documentation is created to explain what the software product does. This way, developers and stakeholders can easily follow the purpose and the underlying functionality of each file/function/line. Documentation can come not only in the form of a `README.md` but also using code as documentation (to write clear code), diagrams, websites, research papers, videos, and external documentation. Besides being a good programming practice, proper technical documentation improves the efficiency of audits. Less time can be spent understanding the protocol and more time can be put towards auditing which improves the efficiency and overall output of the audit.

The `Polygon Id` team provided two documents to assist the audit process, describing (a) the contract `StateV2` and the `SMT` library; (b) instructions for running the test suite. These documents covered the most common terms used in the source code, explanations for the core business logic and functions flow and guides for running tests. Besides that, the audit team used the Polygon public documentation available at `wiki.polygon.technology` to gain deeper knowledge about the `Polygon Id` and understand the interaction between the `Polygon Id` and the `Iden3` protocol. By reading the whole documentation suite, we could get a proper understanding of how the contract `StateV2` should operate.

Along this investigation, we could notice that the implementation for leaf node hash calculations differs from the specification. According to the sparse Merkle Tree specification designed by the iden3 team, a leaf node hash is calculated as follows:

```
////////////////////////////////////////////////////////////////////////////////
// @audit Code snippet 1: how the hash is defined in the iden3 documentation
////////////////////////////////////////////////////////////////////////////////
hash(leaf_bit, key, value)
```

However, the function `getNodeHash(...)` computes the hash as follows:

```
////////////////////////////////////////////////////////////////////////////////
// @audit Code snippet 2: how Polygon is computing the hash
////////////////////////////////////////////////////////////////////////////////
hash(key, value, leaf_bit)
```

After presenting this issue to the Polygon team, we were told that the documentation of the `iden3` is not reflecting the implementation and this is the reason they are adopting the `code snippet 2` presented above. The Polygon Wiki is a complete source of resources for developers and auditors, and we are satisfied with the documentation provided by the Polygon team. The codebase has sufficient inline comments, helping the audit team to understand the functions flow and to detect some issues. The `Polygon Id` could benefit from clearly stating each use case, allowing the audit team to ensure that all use cases are perfectly implemented.

13 Test Suite Evaluation

13.1 Contracts Compilation Output

```
$ npx hardhat compile
contracts/lib/BabyJubJub.sol:10:5: Warning: Function state mutability can be restricted to pure
    function modinv(uint256 a, uint256 q) internal view returns (uint256) {
    ^ (Relevant source part starts here and spans across multiple lines).

Warning: SPDX license identifier not provided in source file. Before publishing, consider adding a comment containing
 ↪  "SPDX-License-Identifier: <SPDX-License>" to each source file. Use "SPDX-License-Identifier: UNLICENSED" for
 ↪  non-open-source code. Please see https://spdx.org for more information.
--> contracts/interfaces/ICircuitValidator.sol


Warning: SPDX license identifier not provided in source file. Before publishing, consider adding a comment containing
 ↪  "SPDX-License-Identifier: <SPDX-License>" to each source file. Use "SPDX-License-Identifier: UNLICENSED" for
 ↪  non-open-source code. Please see https://spdx.org for more information.
--> contracts/interfaces/IERC20zkp.sol


Warning: SPDX license identifier not provided in source file. Before publishing, consider adding a comment containing
 ↪  "SPDX-License-Identifier: <SPDX-License>" to each source file. Use "SPDX-License-Identifier: UNLICENSED" for
 ↪  non-open-source code. Please see https://spdx.org for more information.
--> contracts/interfaces/IState.sol


Warning: SPDX license identifier not provided in source file. Before publishing, consider adding a comment containing
 ↪  "SPDX-License-Identifier: <SPDX-License>" to each source file. Use "SPDX-License-Identifier: UNLICENSED" for
 ↪  non-open-source code. Please see https://spdx.org for more information.
--> contracts/interfaces/IVerifier.sol


Warning: SPDX license identifier not provided in source file. Before publishing, consider adding a comment containing
 ↪  "SPDX-License-Identifier: <SPDX-License>" to each source file. Use "SPDX-License-Identifier: UNLICENSED" for
 ↪  non-open-source code. Please see https://spdx.org for more information.
--> contracts/interfaces/IZKPAirdrop.sol


Warning: SPDX license identifier not provided in source file. Before publishing, consider adding a comment containing
 ↪  "SPDX-License-Identifier: <SPDX-License>" to each source file. Use "SPDX-License-Identifier: UNLICENSED" for
 ↪  non-open-source code. Please see https://spdx.org for more information.
--> contracts/interfaces/IZKPVerifier.sol


Generating typings for: 48 artifacts in dir: typechain for target: ethers-v5
Successfully generated 79 typings!
Compiled 36 Solidity files successfully
```

13.2 Tests Output

```
$ npx hardhat test test/state/stateV2.test.ts test/smt/smt.test.ts
No need to generate any newer typings.


  State transitions positive cases
Warning: Potentially unsafe deployment of StateV2

    You are using the `unsafeAllow.external-library-linking` flag to include external libraries.
    Make sure you have manually checked that the linked libraries are upgrade safe.

     Initial state publishing (1461ms)
     Subsequent state update (677ms)

  State transition negative cases
Warning: Potentially unsafe deployment of StateV2

    You are using the `unsafeAllow.external-library-linking` flag to include external libraries.
    Make sure you have manually checked that the linked libraries are upgrade safe.

     Old state does not match the latest state (636ms)
Warning: Potentially unsafe deployment of StateV2

    You are using the `unsafeAllow.external-library-linking` flag to include external libraries.
    Make sure you have manually checked that the linked libraries are upgrade safe.

     Old state is genesis but identity already exists (630ms)
Warning: Potentially unsafe deployment of StateV2

    You are using the `unsafeAllow.external-library-linking` flag to include external libraries.
    Make sure you have manually checked that the linked libraries are upgrade safe.

++++++++++++++++
VM Exception while processing transaction: reverted with reason string 'Genesis state already exists'
     Genesis state already exists (610ms)
Warning: Potentially unsafe deployment of StateV2

    You are using the `unsafeAllow.external-library-linking` flag to include external libraries.
    Make sure you have manually checked that the linked libraries are upgrade safe.

     New state should not exist (621ms)
Warning: Potentially unsafe deployment of StateV2

    You are using the `unsafeAllow.external-library-linking` flag to include external libraries.
    Make sure you have manually checked that the linked libraries are upgrade safe.

     Old state is not genesis but identity does not yet exist
Warning: Potentially unsafe deployment of StateV2

    You are using the `unsafeAllow.external-library-linking` flag to include external libraries.
    Make sure you have manually checked that the linked libraries are upgrade safe.

     Zero-knowledge proof of state transition is not valid (546ms)

  State history
Warning: Potentially unsafe deployment of StateV2

    You are using the `unsafeAllow.external-library-linking` flag to include external libraries.
    Make sure you have manually checked that the linked libraries are upgrade safe.

[
  BigNumber {
    _hex: '0x102f1530a708f08fe0cfc4dd2486fb8e93a2b6add1128bfcf2a13de66d1202',
    _isBigNumber: true
  },
  BigNumber {
    _hex: '0x05b972ee2ae0c56d75841a4558fefe63ca81c0d8ed9709d687443a610a4686d5',
    _isBigNumber: true
  },
```

```
  BigNumber { _hex: '0x00', _isBigNumber: true },
  BigNumber { _hex: '0x63d1b987', _isBigNumber: true },
  BigNumber { _hex: '0x00', _isBigNumber: true },
  BigNumber { _hex: '0x47', _isBigNumber: true },
  BigNumber { _hex: '0x00', _isBigNumber: true },
  id: BigNumber {
    _hex: '0x102f1530a708f08fe0cfc4dd2486fb8e93a2b6add1128bfcf2a13de66d1202',
    _isBigNumber: true
  },
  state: BigNumber {
    _hex: '0x05b972ee2ae0c56d75841a4558fefe63ca81c0d8ed9709d687443a610a4686d5',
    _isBigNumber: true
  },
  replacedByState: BigNumber { _hex: '0x00', _isBigNumber: true },
  createdAtTimestamp: BigNumber { _hex: '0x63d1b987', _isBigNumber: true },
  replacedAtTimestamp: BigNumber { _hex: '0x00', _isBigNumber: true },
  createdAtBlock: BigNumber { _hex: '0x47', _isBigNumber: true },
  replacedAtBlock: BigNumber { _hex: '0x00', _isBigNumber: true }
]
[
  {
    oldState: '958453131201158201170444404525680013891001007445512785045540173847773513623 5',
    newState: '423022603543755660527105734759826877620384657072717467376478493801710809 4097',
    id: '285945063973374968303362307152175461671493325519456453289582267967773783 06',
    blockNumber: 70,
    timestamp: 1674688902
  },
  {
    oldState: '42302260354375566052710573475982687762038465707271746737647849380171080940 97',
    newState: '25892241699667557081332516592316626926726421244455984702287670144705289602 13',
    id: '285945063973374968303362307152175461671493325519456453289582267967773783 06',
    blockNumber: 71,
    timestamp: 1674688903
  }
]
     should return state history (46ms)
     should be reverted if length is zero
     should be reverted if length limit exceeded
     should be reverted if out of bounds

  get StateInfo negative cases
Warning: Potentially unsafe deployment of StateV2

    You are using the `unsafeAllow.external-library-linking` flag to include external libraries.
    Make sure you have manually checked that the linked libraries are upgrade safe.

     getStateInfoByID: should be reverted if identity does not exist
     getStateInfoHistoryById: should be reverted if identity does not exist
     getStateInfoHistoryLengthById: should be reverted if identity does not exist
     getStateInfoByState: should be reverted if state does not exist

  GIST proofs
Warning: Potentially unsafe deployment of StateV2

    You are using the `unsafeAllow.external-library-linking` flag to include external libraries.
    Make sure you have manually checked that the linked libraries are upgrade safe.

root history length:  BigNumber { _hex: '0x02', _isBigNumber: true }
     Should be correct historical proof by root and the latest root (1421ms)
Warning: Potentially unsafe deployment of StateV2

    You are using the `unsafeAllow.external-library-linking` flag to include external libraries.
    Make sure you have manually checked that the linked libraries are upgrade safe.

[
  BigNumber {
    _hex: '0x03196d5391d63ecca43a580627490a54c39751b935cd0f9b22b9f289c743216c',
    _isBigNumber: true
  },
  BigNumber {
    _hex: '0x07c073f867dc049a93195c8963568e6de7e22b6643d8acb87c7c725108e08988',
    _isBigNumber: true
  },
```

```
  BigNumber { _hex: '0x63d1b9a1', _isBigNumber: true },
  BigNumber { _hex: '0x63d1b9a2', _isBigNumber: true },
  BigNumber { _hex: '0x61', _isBigNumber: true },
  BigNumber { _hex: '0x62', _isBigNumber: true },
  root: BigNumber {
    _hex: '0x03196d5391d63ecca43a580627490a54c39751b935cd0f9b22b9f289c743216c',
    _isBigNumber: true
  },
  replacedByRoot: BigNumber {
    _hex: '0x07c073f867dc049a93195c8963568e6de7e22b6643d8acb87c7c725108e08988',
    _isBigNumber: true
  },
  createdAtTimestamp: BigNumber { _hex: '0x63d1b9a1', _isBigNumber: true },
  replacedAtTimestamp: BigNumber { _hex: '0x63d1b9a2', _isBigNumber: true },
  createdAtBlock: BigNumber { _hex: '0x61', _isBigNumber: true },
  replacedAtBlock: BigNumber { _hex: '0x62', _isBigNumber: true }
]
[
  BigNumber {
    _hex: '0x07c073f867dc049a93195c8963568e6de7e22b6643d8acb87c7c725108e08988',
    _isBigNumber: true
  },
  BigNumber { _hex: '0x00', _isBigNumber: true },
  BigNumber { _hex: '0x63d1b9a2', _isBigNumber: true },
  BigNumber { _hex: '0x00', _isBigNumber: true },
  BigNumber { _hex: '0x62', _isBigNumber: true },
  BigNumber { _hex: '0x00', _isBigNumber: true },
  root: BigNumber {
    _hex: '0x07c073f867dc049a93195c8963568e6de7e22b6643d8acb87c7c725108e08988',
    _isBigNumber: true
  },
  replacedByRoot: BigNumber { _hex: '0x00', _isBigNumber: true },
  createdAtTimestamp: BigNumber { _hex: '0x63d1b9a2', _isBigNumber: true },
  replacedAtTimestamp: BigNumber { _hex: '0x00', _isBigNumber: true },
  createdAtBlock: BigNumber { _hex: '0x62', _isBigNumber: true },
  replacedAtBlock: BigNumber { _hex: '0x00', _isBigNumber: true }
]
     Should be correct historical proof by time (1365ms)
Warning: Potentially unsafe deployment of StateV2

    You are using the `unsafeAllow.external-library-linking` flag to include external libraries.
    Make sure you have manually checked that the linked libraries are upgrade safe.

     Should be correct historical proof by block (1326ms)

  GIST root history
Warning: Potentially unsafe deployment of StateV2

    You are using the `unsafeAllow.external-library-linking` flag to include external libraries.
    Make sure you have manually checked that the linked libraries are upgrade safe.

     Should search by block and by time return same root (1340ms)
Warning: Potentially unsafe deployment of StateV2

    You are using the `unsafeAllow.external-library-linking` flag to include external libraries.
    Make sure you have manually checked that the linked libraries are upgrade safe.

     Should have correct GIST root transitions info (1267ms)

  SMT tests
    Merkle tree proofs of SMT
      SMT existence proof
        add 1 leaf and generate the proof for it (74ms)
        add 2 leaves (depth = 2) and generate the proof of the second one (214ms)
        add 2 leaves (depth = 2) update 2nd one and generate the proof of the first one (299ms)
        add 2 leaves (depth = 2) update the 2nd leaf and generate the proof of the second one (284ms)
        add 2 leaves (depth = 2) update the 2nd leaf and generate the proof of the first one for the previous root state
        ↪  (290ms)
        add 2 leaves (depth = 2) update the 2nd leaf and generate the proof of the second one for the previous root
        ↪  state (291ms)
```

```
      SMT non existence proof
         add 1 leaf and generate a proof on non-existing leaf (69ms)
         add 2 leaves (depth = 2) and generate proof on non-existing leaf WITH aux node (214ms)
         add 2 leaves (depth = 2) and generate proof on non-existing leaf WITHOUT aux node (202ms)
         add 2 leaves (depth = 2), update the 2nd leaf and generate proof of non-existing leaf WITH aux node (which
         ↪  existed before update) (296ms)
         add 2 leaves (depth = 2), update the 2nd leaf and generate proof of non-existing leaf WITHOUT aux node (312ms)
         add 2 leaves (depth = 2), add 3rd leaf and generate proof of non-existance for the 3rd leaf in the previous
         ↪  root state (275ms)
      SMT add leaf edge cases
         Positive: add two leaves with maximum depth (882ms)
         Negative: add two leaves with maximum depth + 1 (86ms)
    Root history requests
Warning: Potentially unsafe deployment of StateV2

    You are using the `unsafeAllow.external-library-linking` flag to include external libraries.
    Make sure you have manually checked that the linked libraries are upgrade safe.

       should return the root history (46ms)
       should revert if length is zero
       should be reverted if length limit exceeded
       should be reverted if out of bounds
    Binary search in SMT root history
      Empty history
         Should return zero root for some search
      One root in the root history
         Should return the first root when equal
         Should return zero when search for less than the first
         Should return the last root when search for greater than the last
      Two roots in the root history
         Should return the first root when search for equal
         Should return the second root when search for equal
         Should return zero when search for less than the first
         Should return the last root when search for greater than the last
      Three roots in the root history
         Should return the first root when equal (39ms)
         Should return the second root when equal
         Should return the third root when equal (44ms)
         Should return zero root when search for less than the first (40ms)
         Should return the last root when search for greater than the last (42ms)
      Four roots in the root history
         Should return the first root when equal (50ms)
         Should return the fourth root when equal (47ms)
         Should return zero when search for less than the first (49ms)
         Should return the last root when search for greater than the last (51ms)
      Search in between the values
         Should return the first root when search in between the first and second (57ms)
         Should return the fourth root when search in between the fourth and the fifth (58ms)
      Search in array with duplicated values
         Should return the last root among two equal values when search for the value (63ms)
         Should return the last root among three equal values when search for the value (56ms)
      Search in array with duplicated values and in between values
         Should search in between the third (1st, 2nd, 3rd equal) and fourth values and return the third (89ms)
         Should search in between the fifth (4th, 5th equal) and sixth values and return the fifth (79ms)
    Edge cases with exceptions
       getRootInfo() should throw when root does not exist (66ms)
       getProofByRoot() should throw when root does not exist (80ms)
       add() should throw when node already exist with the same index and value (119ms)


  65 passing (28s)
```

13.3 Code Coverage

```
npx hardhat coverage
```

The relevant output is presented below.

```
-------------------------------------------|----------|----------|----------|----------|----------------|
File                                       |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
-------------------------------------------|----------|----------|----------|----------|----------------|
 contracts/state/StateV2.sol               |    91.84 |    96.15 |    82.61 |    92.31 |135,224,419,432 |
 contracts/lib/Smt.sol                     |    96.43 |    84.85 |       95 |    96.38 |... 406,509,611 |
-------------------------------------------|----------|----------|----------|----------|----------------|
All files                                  |    94.14 |    90.50 |    88.80 |    94.35 |                |
-------------------------------------------|----------|----------|----------|----------|----------------|
```

13.4 Slither

All the relevant issues raised by Slither have been incorporated into the issues described in this report.

14 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.