Security Review Report
NM-0078 WORLDCOIN

NETHERMIND

(May 22, 2023)

Contents

1 Executive Summary

This document presents the security review performed by Nethermind on the World ID contracts. **World ID** is a privacy-first identity protocol that brings global proof of personhood to the internet. It enables verified users to prove they are unique individuals and, optionally, demonstrate that they perform a given action only once. This is accomplished with Blockchain technology and zero-knowledge proofs (ZKP).

The audited code consists of three different components listed below:

- World ID contracts containing 700 lines of code with 89.22% of code coverage;
- World ID State Bridge contracts containing 232 lines of code with 32.60% of code coverage; and
- World ID Example Airdrop contracts containing 123 lines of code with 100% of code coverage.

Each component includes a **README** file that briefly introduces the code and instructions on using, testing, and deploying these contracts. Moreover, official documentation was utilized during this assessment. The review was supported by open communication with the development team, allowing for a more comprehensive understanding of the protocol mechanisms and addressing any potential gaps.

The audit was performed using (a) manual analysis of the codebase, (b) automated analysis tools, (c) simulation of the smart contracts, and (d) creation of test cases. **Along this document, we report** 15 points of attention, where two are classified as `Critical`, two are classified as `Medium`, four are classified as `Low`, six are classified as `Informational` or `Best Practices`, and one is classified as `Undetermined`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope of this audit. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology adopted for this audit. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the compilation, tests, and automated tests. Section 9 concludes the document.

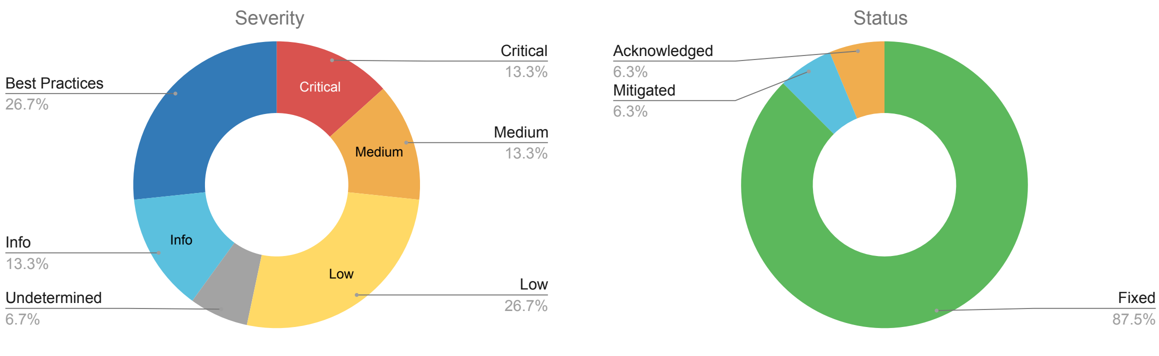

Fig. 1: Distribution of issues: Critical (2), **High** (0), **Medium** (2), **Low** (4), **Undetermined** (1), **Informational** (2), **Best Practices** (4).
Distribution of status: Fixed (13), **Acknowledged** (1), **Mitigated** (1), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Apr. 17, 2023
Response from Client	May 01, 2023
Final Report	May 22, 2023
Methods	Manual Review, Automated Analysis
Repository	WorldID, State Bridge, Airdrop Example
Commit Hash (Initial Audit - WorldID)	20fc82fb52125bf33eb150d45c096441ae516296
Commit Hash (Initial Audit - State Bridge)	6d01d2178d935c9aa669cf239888f8b1603ee63b
Commit Hash (Initial Audit - Airdrop Example)	1150365d420498899efce5d4d5b3f83daa85f98d
Documentation	WorldID Docs WorldId - README State Bridge - README Airdrop Example - README
Documentation Assessment	High
Test Suite Assessment	High

2 Audited Files

2.1 World ID

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/WorldIDIdentityManager.sol	6	24	400.0%	4	34
2	src/WorldIDRouter.sol	6	22	366.7%	4	32
3	src/WorldIDRouterImplV1.sol	121	170	140.5%	45	336
4	src/WorldIDIdentityManagerImplV1.sol	431	437	101.4%	104	972
5	src/abstract/WorldIDProxy.sol	6	19	316.7%	3	28
6	src/abstract/WorldIDImpl.sol	14	42	300.0%	6	62
7	src/utils/CheckInitialized.sol	16	17	106.2%	6	39
8	src/utils/UnimplementedTreeVerifier.sol	17	19	111.8%	3	39
9	src/utils/SemaphoreTreeDepthValidator.sol	8	8	100.0%	1	17
10	src/data/VerifierLookupTable.sol	41	74	180.5%	20	135
11	src/interfaces/IWorldID.sol	10	16	160.0%	1	27
12	src/interfaces/IBridge.sol	4	4	100.0%	1	9
13	src/interfaces/IWorldIDGroups.sol	11	18	163.6%	1	30
14	src/interfaces/ITreeVerifier.sol	9	16	177.8%	1	26
	Total	**700**	**886**	**126.6%**	**200**	**1786**

2.2 State Bridge

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/StateBridge.sol	74	72	97.3%	32	178
2	src/OpWorldID.sol	16	39	243.8%	8	63
3	src/PolygonWorldID.sol	28	40	142.9%	10	78
4	src/abstract/WorldIDBridge.sol	76	109	143.4%	39	224
5	src/utils/SemaphoreTreeDepthValidator.sol	8	7	87.5%	1	16
6	src/interfaces/ICrossDomainOwnable3.sol	4	9	225.0%	1	14
7	src/interfaces/IWorldID.sol	10	17	170.0%	1	28
8	src/interfaces/IWorldIDIdentityManager.sol	4	9	225.0%	1	14
9	src/interfaces/IBridge.sol	4	7	175.0%	1	12
10	src/interfaces/ISendBridge.sol	4	7	175.0%	1	12
11	src/interfaces/IOpWorldID.sol	4	10	250.0%	1	15
	Total	**232**	**326**	**140.5%**	**96**	**654**

2.3 Example Airdrop

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/WorldIDAirdrop.sol	56	53	94.6%	26	135
2	src/WorldIDMultiAirdrop.sol	67	61	91.0%	31	159
	Total	**123**	**114**	**92.7%**	**57**	**294**

3 Summary of Issues

	Finding	Severity	Update
1	Airdrops' funds can be stolen from the `WorldIDMultiAirdrop` contract	Critical	Fixed
2	Old roots can allow invalid proofs to be submitted as valid in the contract `WorldIdBridge`	Critical	Fixed
3	Centralization Risks	Medium	Mitigated
4	`PolygonWorldID` sender and receiver can be set by any address	Medium	Fixed
5	Absence of `ROOT_HISTORY_EXPIRY` update mechanism in `StateBridge` contract	Low	Fixed
6	Lack of a two-step process for transferring ownership	Low	Fixed
7	`NonExistentRoot()` may not be executed for nonexistent roots	Low	Fixed
8	`actionId` is hashed two times in `WorldIDAirdrop` contract	Low	Fixed
9	`ActionID` may not be unique across applications	Undetermined	Acknowledged
10	Double initialization of the tree depth	Info	Fixed
11	Unnecessary storage writes	Info	Fixed
12	Lack of events for relevant operations	Best Practices	Fixed
13	Low-level calls may cause errors	Best Practices	Fixed
14	Non-private functions are not virtual	Best Practices	Fixed
15	Presence of unused variables	Best Practices	Fixed

4 System Overview

The audit encompasses three components: a) `World ID contracts`, comprising fourteen contracts; b) `World ID state bridge`, consisting of eleven contracts; and c) `World ID airdrop`, containing two contracts. Fig. 2 illustrates a structural diagram of the contracts and their relationships with `WorldIDIdentityManager`. Subsequently, we outline the primary contracts (highlighted in yellow): `VerifierLookupTable`, `WorldIDIdentityManager`, `WorldIDRouter`, `StateBridge`, `PolygonWorldID`, and `OpWorldID`.

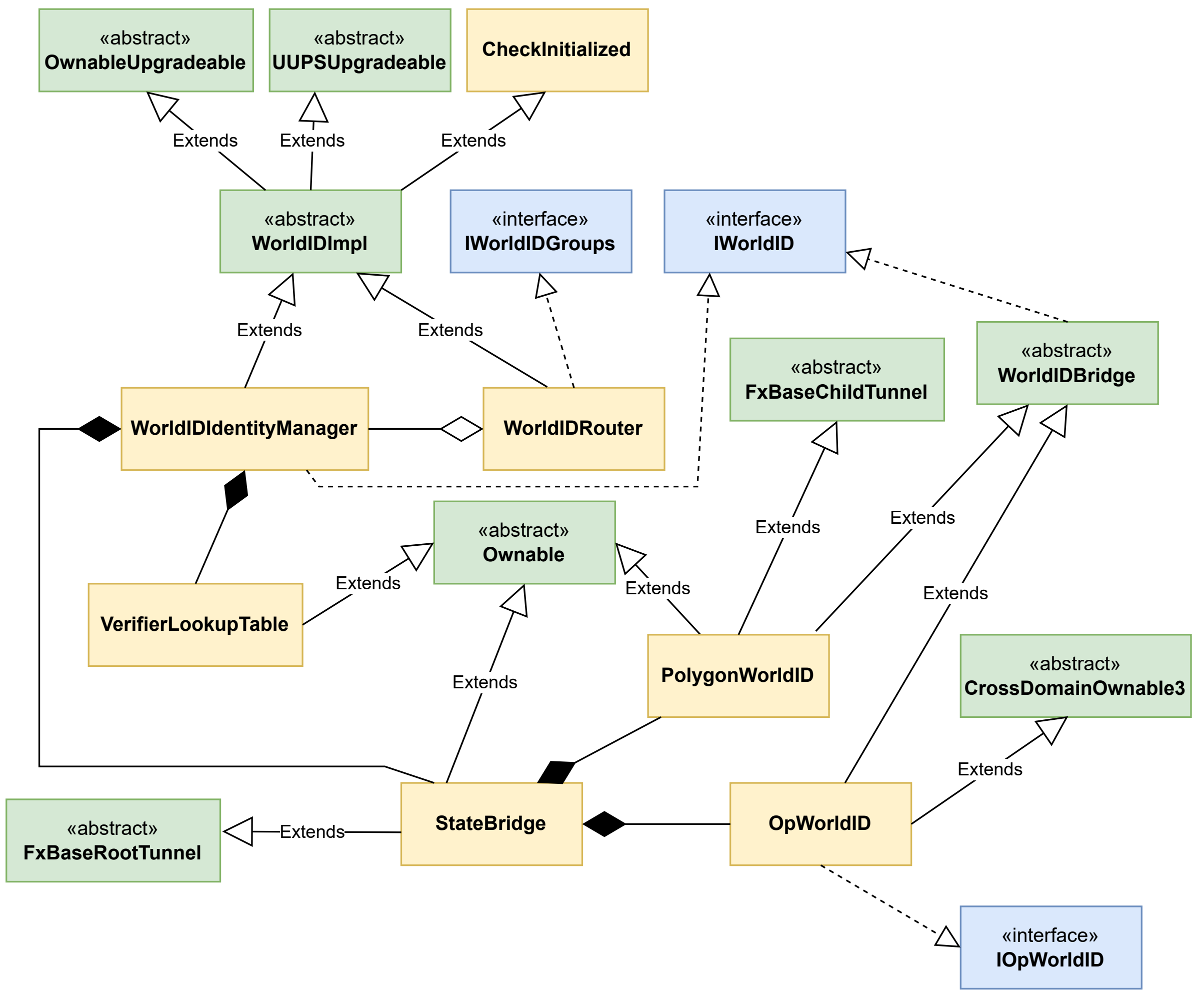

Fig. 2: Class diagram of the main contracts and their relationships.

4.1 World ID Contracts & World ID State Bridge

VerifierLookupTable: This contract establishes a mapping between batch sizes and tree verifiers, returning the suitable verifier for a specific batch size before employing that verifier to validate a tree modification proof. The contract inherits from the `Ownable` contract, which offers a straightforward access control mechanism, ensuring only the contract owner can modify the lookup table. The contract features several public functions for interacting with the lookup table:

a. The `getVerifierFor(...)` function takes a batch size as input and returns the associated tree verifier instance. It checks that the batch size is valid, i.e., that there is an associated verifier for that size.

b. The `addVerifier(...)` function allows the owner to add a new verifier for a given batch size. It checks that there is no existing verifier for the given batch size and returns an error if there is.

c. The `updateVerifier(...)` function allows the owner to update an existing verifier for a given batch size.

d. The `disableVerifier(...)` function allows the owner to disable a verifier for a given batch size by setting it to null.

WorldIDIdentityManager: Designed to operate behind a proxy, this contract implements a batch-based identity manager for the WorldID protocol, which verifies externally created Zero Knowledge Proofs. The contract features various public functions serving different purposes, such as:

a. The functions `setRegisterIdentitiesVerifierLookupTable(...)` and `setIdentityUpdateVerifierLookupTable(...)`, which allow setting lookup tables of verifiers used for identity updates.

b. The function `setSemaphoreVerifier(...)` enables setting the address for the semaphore verifier used for verifying semaphore proofs.

c. The functions `enableStateBridge(...)` and `disableStateBridge(...)`, which enable or disable the state bridge.

d. The function `verifyProof(...)`, which allows verifying semaphore proofs.

e. The `updateIdentities(...)` and `registerIdentities(...)` functions, allowing the owner to modify the current registry of identities in a group. It is worth mentioning that most checks are done when modifying the identities in the group are executed in the proof generation phase and not performed on the smart contracts layer.

The contract owner can only call all of the previously mentioned functions, except for the `verifyProof(...)`. Two of the functions, `registerIdentities(...)` and `updateIdentities(...)`, require meeting specific conditions for successful verification. After meeting these conditions, the latest root is updated, the previous root is added to history, and its expiration timestamp is set. Fig. 3 shows a sequence diagram to illustrate how contracts interact with each other to register identities. As can be observed, `registerIdentities(...)` sends the latest WorldID Identity Manager root to all chains.

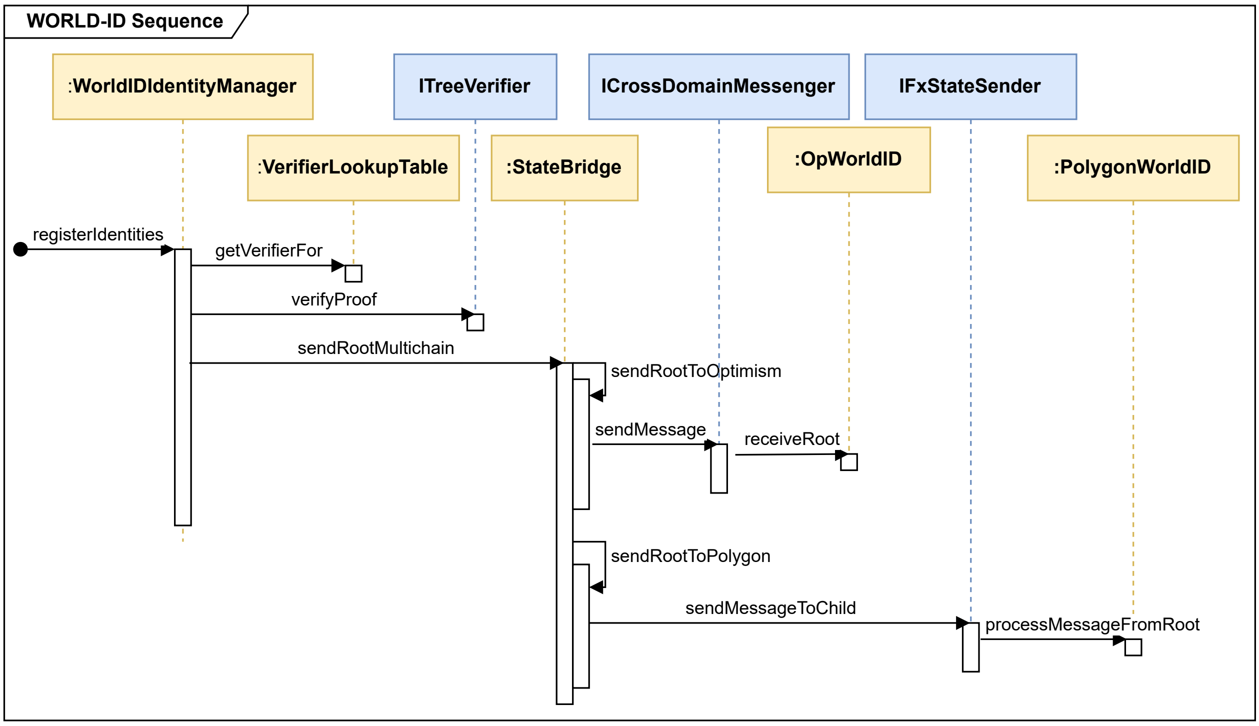

Fig. 3: Sequence diagram for the function `WorldIDIdentityManager.registerIdentities(...)`.

WorldIDRouter: This contract is designed to operate behind a proxy. Specifically, `WorldIDRouter` is responsible for dispatching group numbers to the correct identity manager implementation. The contract implements several public functions that serve different purposes:

a. `routeFor(...)` returns the target address to specific groups based on a provided group number.

b. `addGroup(...)` adds a new group to the router.

c. `updateGroup(...)` updates the target address for a group in the router.

d. `disableGroup(...)` disables the target group in the router.

e. `groupCount(...)` returns the number of groups in the routing table.

StateBridge: The `StateBridge` contract is responsible for distributing new World ID Identity Manager roots and timestamps to the supported chains, namely Polygon and Optimism. This contract lives on Ethereum Mainnet and is called by the `WorldIDIdentityManager` contract through the `registerIdentities(...)` and `updateIdentities(...)` functions described below:

a. `sendRootMultichain(...)` is used to transmit the latest WorldID Identity Manager root to all chains supported by the WorldID Identity Manager contract. The root is provided as an argument to this function. Before transmitting, this function verifies that the caller is the WorldID Identity Manager contract. Fig. 3 describes the sequence to send the root to the chains.

b. `transferOwnershipOptimism(...)` allows the `StateBridge` contract to transfer ownership of `OpWorldID` to another contract on L1 or to a local Optimism EOA. This function can only be called by the contract owner.

PolygonWorldID: The `PolygonWorldID` contract manages the root history of the WorldID Merkle root on Polygon PoS. Its primary goal is to allow semaphore-proof verifications on the Polygon PoS chain. As described in Fig. 2, it inherits from the `WorldIDBridge`, `FxBaseChildTunnel`, and `Ownable` contracts. This contract is deployed on Polygon PoS and is invoked by the `StateBridge` contract for each new root insertion. Fig. 3 illustrates the internal function `StateBridge.sendRootToPolygon` sending the new root to `PolygonWorldID`. This contract implements the following functions:

a. `setRootHistoryExpiry(...)` sets the amount of time it takes for a root in the root history to expire. It takes in one parameter: `expiryTime`, which is the new amount of time it takes for a root to expire. This function can only be called by the contract owner.

b. `_processMessageFromRoot(...)` is an internal override function called via a system call to receive messages from the `StateBridge` contract. It calls the `_receiveRoot` function upon receiving a message from the `StateBridge` contract via the `FxChildTunnel`.

c. The function `verifyProof(...)`, which allows verifying semaphore proofs.

OpWorldID: Similarly to the `PolygonWorldID` contract, `OpWorldID` manages the root history of the WorldID Merkle root, but it is deployed on the Optimism chain. Its primary goal is to allow semaphore-proof verifications on the Optimism chain. As described in Fig. 2, it inherits from the `WorldIDBridge` and `CrossDomainOwnable3` contracts and implements the `IOpWorldID` interface. This contract is invoked by the `StateBridge` contract for each new root insertion. Fig. 3 illustrates the internal function `StateBridge.sendRootToOptimism` sending the new root to `OpWorldID`. This contract implements the following functions:

a. `receiveRoot(...)` is called by the state bridge contract when it forwards a new root to the bridged WorldID. It sets the new root and supersedes the timestamp in the root history.

b. `setRootHistoryExpiry(...)` sets the expiration time for a root in the root history to expire. Only the owner of the contract can call this function.

c. The function `verifyProof(...)`, which allows verifying semaphore proofs.

4.2 World ID AirDrop

This component consists of two template contracts for WorldID users: a) `WorldIDAirdrop` is aimed for airdropping tokens to World ID members, and b) `WorldIDMultiAirdrop` manages multiple airdrops to World ID users. Fig. 4 illustrates the interactions between contracts in the sequential order to claim the airdrop in `WorldIDAirdrop` contract.

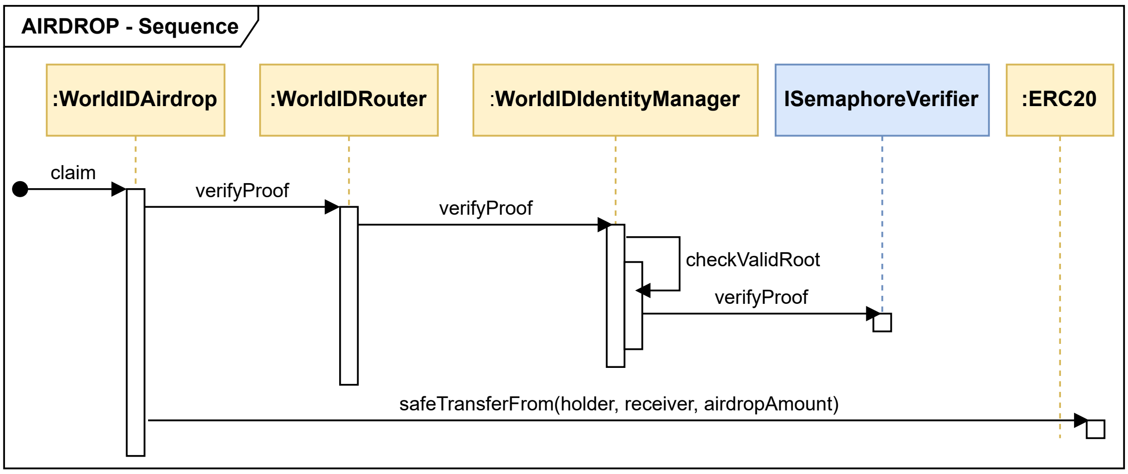

Fig. 4: Sequence diagram for the function `WorldIDAirdrop.claim(...)`.

5 Abstract Formal Specification for Poseidon hash

In this section, we present a formal proof of equivalence between implementing the Poseidon hash and a zero-knowledge circuit for it using the Lean proof assistant. Firstly, we provide a brief overview of the Poseidon hash (Section 5.1), followed by a discussion of the Lean formalization in Section 5.2.

5.1 An overview of Poseidon hash

Let $\mathbb{F}_p$ be a prime field where p is approximately greater than 2^{31}, i.e., $p \approx 2^n > 2^{31}$. The Poseidon hash is a function that maps strings from $\mathbb{F}p$ to fixed-length strings over $\mathbb{F}p^o$, where o is usually equal to 1.Implementing the Poseidon hash uses the concept of *rounds*, executed over the state a number of times. Each round applies an `S-box` function to the entire state (full rounds) or a single value, while the rest remains unchanged (half-rounds). Intuitively, each round function consists of the following three steps:

a. A step for adding constants, which sums a vector to a column of the state representation;

b. A subwords step applies an `S-box` function to the state. When the `S-box` is applied to just one state entry; we call this a half-round. Applying `S-box` to all state entries is named a full round.

c. A mixing step multiplies the current state by an MDS matrix.

The next Section presents more details about the Lean formalization of Poseidon hash and the proof of equivalence with a zero-knowledge circuit using the Lean proof assistant.

5.2 Lean formalization

Here, we present more details about the Poseidon hash implementation in Lean and proof of its equivalence with encoding a `ZK-circuit` as a relational specification. For each implementation component, we show its Lean implementation and its corresponding `ZK-circuit` together with a theorem statement establishing its correctness.

5.2.1 Definition of a finite field type

We start our formalization by defining a type for denoting finite fields:

```
def Order : ℕ := -- some prime number
variable [Fact (Nat.Prime Order)]
abbrev F := ZMod Order
```

We start by defining a constant `Order` for denoting the order of our finite field type. We postulate the fact that `Order` is a prime number and then define a type `F` as the type of integers modulus the prime `Order`.

5.2.2 Function sbox

The first function defined for the Poseidon implementation is `sbox`:

```
def sbox(v : F) : F := v ^ 5
```

The function takes a field value and raises it to the 5th power. The following proposition describes the circuit definition for this function:

```
def sboxConstraint (v : F)(res : F) : Prop :=
  ∃ v₂ v₄, v₂ = v * v ∧ v₄ = v₂ * v₂ ∧ res = v * v₄
```

Finally, we can say that sbox function is sound by stating the following theorem:

```
theorem sboxSound (v res : F) : sboxConstraint v res ↔ sbox v = res
```

The theorem ensures that `sbox` result agrees with the circuit defined by `sboxConstraint`.

5.2.3 Function applyMDS

The next function definition is `applyMDS`, which multiplies the current state by an MDS matrix. Its implementation uses Lean Mathlib's matrix multiplication operation.

```
def applyMDS {n : ℕ} (state : Matrix (Fin (n + 1)) Unit F)
                    (mds : Matrix (Fin (n + 1)) (Fin (n + 1)) F) : Matrix (Fin (n + 1)) Unit F
    := Matrix.mul mds state
```

The circuit for expressing matrix multiplication is defined by quantifying all possible rows of the output as follows:

```
def applyMDSConstraint
    {n : ℕ}
    (mds : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
    (state : Matrix (Fin (n + 1)) Unit F)
    (res : Matrix (Fin (n + 1)) Unit F) : Prop :=
        ∀ i, res i unit = Σ v, (mds i v) * state v unit
```

The soundness theorem for the function applyMDS ensures that the function's result agrees with the circuit implementation:

```
theorem applyMDSSound {n : ℕ} (mds : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                              (state : Matrix (Fin (n + 1)) Unit F)
                              (res : Matrix (Fin (n + 1)) Unit F) :
      applyMDSConstraint mds state res ↔ applyMDS state mds = res
```

The Lean proof script for this theorem can be found here.

5.2.4 Function half-round

The definition of the function half-round takes the current state, the MDS matrix, the index of the column that should be used by the actual round, the matrix of constants, and it performs the following steps:

- Get the column that the current round should operate on;
- Add the previously obtained column to the current state value;
- Apply the sbox function to the first entry of the new state value;
- Apply the MDS matrix to the newly updated state.

```
def halfRound {n : ℕ}(mds : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                 (pround : Fin (n + 1))
                 (constants : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                 (state : Matrix (Fin (n + 1)) Unit F)
         : Matrix (Fin (n + 1)) Unit F :=
        let constm := col (constants pround)
        let state' := state + constm
        let state'' : Matrix (Fin (n + 1)) Unit F := updateEntry state'
      applyMDS state'' mds
```

The relational definition of the ZK-circuit for half-round is:

```
def halfRoundConstraint {n : ℕ}(mds : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                               (pround : Fin (n + 1))
                               (constants : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                               (state : Matrix (Fin (n + 1)) Unit F)
                              (res : Matrix (Fin (n + 1)) Unit F) : Prop :=
        ∃ res₁ res₂,
        sumGadget state (col (constants pround)) res₁ ∧
        updateEntryConstraint res₁ res₂ ∧
        applyMDSConstraint mds res₂ res
```

it uses the sumGadget, the constraint representation for the sum of two vectors. The definition of the sumGadget constraint can be found here. The soundness theorem for the function half-round ensures that the function's result agrees with its circuit implementation:

```
theorem halfRoundSound {n : ℕ}(mds : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                              (pround : Fin (n + 1))
                               (constants : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                               (state : Matrix (Fin (n + 1)) Unit F)
                               (res : Matrix (Fin (n + 1)) Unit F) :
      halfRoundConstraint mds pround constants state res ↔ halfRound mds pround constants state = res
```

The complete Lean proof script for the soundness theorem can be found online here.

5.2.5 Function full-round

The definition of the function `full-round` takes the current state, the MDS matrix, the index of the column that should be used by the actual round, and the matrix of constants, and it performs the following steps:

- Get the column that the current round should operate on;
- Add the previously obtained column to the current state value;
- Apply the sbox function all entries of the current state;
- Apply the MDS matrix to the newly updated state.

```
def fullRound {n : ℕ} (mds : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                        (fround : Fin (n + 1))
                        (constants : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                        (state : Matrix (Fin (n + 1)) Unit F)
            : Matrix (Fin (n + 1)) Unit F :=
        let constm := col (constants fround)
        let state' := state + constm
        let state'' := Matrix.map state' sbox
        applyMDS state'' mds
```

The relational definition of the `ZK-circuit` for `full-round` is:

```
def fullRoundConstraint  {n : ℕ} (mds : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                                 (pround : Fin (n + 1))
                                 (constants : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                                 (state : Matrix (Fin (n + 1)) Unit F)
                                 (res : Matrix (Fin (n + 1)) Unit F) : Prop :=
           ∃ res₁, sumGadget state (col (constants pround)) res₁ ∧
           ∃ res₂, (∀ i, sboxConstraint (res₁ i unit) (res₂ i unit)) ∧
           applyMDSConstraint mds res₂ res
```

The soundness theorem for the function `full-round` ensures that the function's result agrees with its circuit implementation:

```
theorem fullRoundSound {n : ℕ} (mds : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                               (pround : Fin (n + 1))
                               (constants : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                               (state : Matrix (Fin (n + 1)) Unit F)
                               (res : Matrix (Fin (n + 1)) Unit F) :
       fullRoundConstraint mds pround constants state res ↔ fullRound mds pround constants state = res
```

The complete Lean proof script for the soundness theorem can be found online here.

5.2.6 Function poseidonIter

The function `poseidonIter` implements the core functionality of the Poseidon hashing by iterating `nTotal` times over the state. Based on the current iteration, the algorithm decides if it should apply a half or full round to the current state. After `nTotal` iterations the function `poseidonIter` returns the resulting state matrix.

```
def poseidonIter {n : ℕ}(mds : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                        (nTotal nFull : ℕ)
                        (constants : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                        (state : Matrix (Fin (n + 1)) Unit F)
      : ∀ (i : ℕ), Matrix (Fin (n + 1)) Unit F :=
      λ i =>
        if h : i < nTotal + 1 then
           let state' := if i < nFull || i < (nTotal - nFull) then
                                        fullRound mds i constants state
                                     else halfRound mds i constants state
           poseidonIter mds nTotal nFull constants  state' (i + 1)
        else state
termination_by poseidonIter mds nTotal nFull constants state i => nTotal + 1 - i
decreasing_by {apply minus_lt ; exact h}
```

The clauses termination_by and decreasing_by are used by the Lean equation compiler to guarantee the poseidonIter does terminate. The ZK-circuit for poseidonIter is defined as:

```
def poseidonIterConstraint {n : ℕ}(mds : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                                  (nTotal nFull : ℕ)
                                  (constants : Matrix (Fin (n + 1)) (Fin (n + 1)) F) :
                                  Matrix (Fin (n + 1)) Unit F →
                                  Matrix (Fin (n + 1)) Unit F →  ℕ → Prop :=
     λ state res i =>
         if h : i < nTotal + 1 then
           if i < nFull ∨ i < (nTotal - nFull) then
                ∃ res₁, fullRoundConstraint mds i constants state res₁ ∧
                poseidonIterConstraint mds nTotal nFull constants res₁ res (i + 1)
           else
                ∃ res₁, halfRoundConstraint mds i constants state res₁ ∧
                poseidonIterConstraint mds  nTotal nFull constants res₁ res (i + 1)
         else res = state
termination_by poseidonIterConstraint _ nTotal _ _ _ _ i => nTotal + 1 - i
decreasing_by {apply minus_lt ; exact h}
```

The soundness proof for poseidonIterConstraint ensures that the functions result agrees with its circuit implementation:

```
theorem poseidonIterSound  {n : ℕ}
                           (mds : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                           (nTotal nFull : ℕ)
                           (constants : Matrix (Fin (n + 1)) (Fin (n + 1)) F) :
                           ∀ state res i,
        poseidonIterConstraint mds nTotal nFull constants state res i ↔
        poseidonIter mds nTotal nFull constants state i = res :=
```

The complete proof script for poseidonIterSound can be found at here.

5.2.7 Function poseidon

The function poseidon implements the hashing algorithm by getting the updated state by poseidonIter and returns the first entry of the resulting matrix.

```
def poseidon {n : ℕ}(mds : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                    (nTotal nFull : ℕ)
                    (constants : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
                    (state : Matrix (Fin (n + 1)) Unit F) : F :=
      let state' := poseidonIter mds nTotal nFull constants state 0
      state' 0 unit
```

The relational definition of the ZK-circuit for poseidon is:

```
def poseidonConstraint
    {n : ℕ}
    (mds : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
    (nTotal nFull : ℕ)
    (constants : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
    (state : Matrix (Fin (n + 1)) Unit F)(res : F) : Prop :=
    ∃ res₁, poseidonIterConstraint mds nTotal nFull constants state res₁ 0 ∧
                  res = res₁ 0 unit
```

The soundness theorem for the function poseidon ensures that the function's result agrees with its circuit implementation:

```
theorem poseidonSound
    {n : ℕ}
    (mds : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
    (nTotal nFull : ℕ)
    (constants : Matrix (Fin (n + 1)) (Fin (n + 1)) F)
    (state : Matrix (Fin (n + 1)) Unit F)
    (res : F) :
      poseidonConstraint mds nTotal nFull constants state res ↔
      poseidon mds nTotal nFull constants state = res
```

The complete Lean proof script for the soundness theorem for poseidon can be found online here.

5.3 Final Remarks

We presented an abstract formal specification for the Poseidon hash function as `ZK-circuits` using the Lean proof assistant. The formalization is composed of three main components: 1) a specification of the Poseidon hash algorithm as Lean functions; 2) the implementation of Poseidon hash as `ZK-circuits` implemented as Lean propositions over prime fields and 3) proofs ensuring the soundness of the circuit representations with respect to the Poseidon hash specification. The complete Lean formalization is available at https://github.com/NethermindEth/WorldCoin-poseidon-verification.

6 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 measures how likely an attacker will uncover and exploit the finding. 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 an attacker exploited the finding. 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.

7 Issues

7.1 [Critical] Airdrops' funds can be stolen from the `WorldIDMultiAirdrop` contract

File(s): `WorldIDMultiAirdrop.sol`

Description: The `WorldIDMultiAidrop` contract allows any user to create a new airdrop by calling the `createAirdrop(...)` function:

```
function createAirdrop(uint256 groupId, ERC20 token, address holder, uint256 amount) public {
    Airdrop memory airdrop = Airdrop({
        groupId: groupId,
        token: token,
        manager: msg.sender,
        ///////////////////////////////////////////////////////////////////////////
        // @audit "holder" argument can be any address
        ///////////////////////////////////////////////////////////////////////////
        holder: holder,
        amount: amount
    });

    getAirdrop[nextAirdropId] = airdrop;
    emit AirdropCreated(nextAirdropId, airdrop);

    ++nextAirdropId;
}
```

To create an airdrop, the creator needs to provide four parameters:

- `groupId`: The WorldID group containing all eligible recipients for the airdrop;
- `token`: The asset to be airdropped;
- `holder`: The account holding the funds to be airdropped. This account must approve sufficient funds for the `WorldIDMultiAirdrop`;
- `amount`: The amount sent to each user claiming the airdrop;

When a user claims their reward for a specific airdrop, they provide the airdrop's `id` and data proving their membership in the eligible group. If the proof is valid, the `WorldIDMultiAirdrop` contract transfers the reward from the airdrop's `holder` to the `receiver`.

```
worldIdRouter.verifyProof(
    root,
    airdrop.groupId,
    abi.encodePacked(receiver).hashToField(),
    nullifierHash,
    abi.encodePacked(address(this), airdropId).hashToField(),
    proof
);

nullifierHashes[nullifierHash] = true;
emit AirdropClaimed(airdropId, receiver);
///////////////////////////////////////////////////////////////////////////
// @audit - Assets are sent from the "holder" to the "receiver" parameter
///////////////////////////////////////////////////////////////////////////
SafeTransferLib.safeTransferFrom(airdrop.token, airdrop.holder, receiver, airdrop.amount);
```

As there are no restrictions on the `holder` parameter during airdrop creation, a malicious user can create a new airdrop using any address that has granted an allowance to the `WorldIDMultiAirdrop` contract as the `holder`. The attacker can then claim the airdrop, effectively stealing assets from the `holder`'s account.

Recommendation(s): Create separate contracts for each airdrop to ensure that the `holder` only grants allowance to the specific airdrop.

Status: Fixed.

Update from client: We decided that we won't use WorldIDMultiAirdrop.sol in production anymore and we have deleted the contract from the repo. Fixed in commit world-id-example-airdrop@b830bf8.

7.2 [Critical] Old roots can allow invalid proofs to be submitted as valid in the contract `WorldIdBridge`

File(s): `WorldIDBridge.sol`

Description: The `WorldIdBridge.verifyProof(...)` function enables verification of the proof provider's group membership. As groups are subject to change over time, the validity of roots endures for some time after their replacement. To verify the validity of a given root at the present time, the `checkValidRoot(...)` function is employed. This function is intended to return true for valid roots and to revert for invalid ones. The function with audit comments is reproduced below.

```
function checkValidRoot(...) public view returns (bool isValid) {
    if (root == _latestRoot) {
        return true;
    }

    uint128 rootTimestamp = rootHistory[root];

    if (block.timestamp - rootTimestamp > ROOT_HISTORY_EXPIRY) {
       revert ExpiredRoot();
    }

    if (rootTimestamp == 0) {
       revert NonExistentRoot();
    }
    ////////////////////////////////////////////////////
    // @audit When the old block is still valid
    //         the value of `isValid` is false
    ////////////////////////////////////////////////////
}
```

The function `verifyProof(...)` is expected to revert when invalid proofs are provided. The function is shown in the next code snippet.

```
function verifyProof(...) public view virtual onlyProxy onlyInitialized {
    /////////////////////////////////////////////
    // @audit when `checkValidRoot` returns false
    //         the proof is not verified by
    //         `semaphoreVerifier`
    /////////////////////////////////////////////
    if (checkValidRoot(root)) {
        semaphoreVerifier.verifyProof( root, nullifierHash, signalHash, externalNullifierHash, proof, treeDepth );
    }
}
```

Nevertheless, it is evident from the aforementioned code that the `checkValidRoot(...)` function produces a `false` return value for older, albeit still valid, roots, which is an erroneous outcome. This particular flaw carries a greater risk, given that the `verifyProof(...)` function does not execute the `verifyProof(...)` function from the semaphore contract in instances where the outcome of `checkValidRoot(...)` is `false`. As a result, `verifyProof(...)` will not revert even in the case of an invalid proof, which may enable an attacker to submit invalid proofs as valid when an old root remains valid.

Recommendation(s): Add the `return` statement in the `checkValidRoot(...)` function with the proper value when the `root` is still valid.

Status: Fixed.

Update from the client: Fixed in commits `world-id-state-bridge@5a340f6`, world-id-state-bridge@1ad9dc2

7.3 [Medium] Centralization Risks

Description: The reviewed contracts contain multiple sensitive actions that can only be executed by accounts with special roles. These actions include:

- Upgrading the logic executed by the system;
- Managing the roots representing identities in each group;
- Modifying the verifier contracts utilized for semaphore and group change proofs;
- Handling data sent to bridges;
- Altering expiration times for roots;
- Managing the routing table for accessing the appropriate WorldID contract;
- Managing contracts' ownership;

These actions could render the system unusable or damage its users. In the event of private key leaks or similar incidents, malicious actors could easily harm the protocol and its users.

Recommendation(s): To mitigate this issue, consider implementing the following measures:

- Utilize a multi-signature wallet as the holder of any privileged role;
- Implement a time lock for any sensitive action, allowing users and developers to intervene in case of malicious intent to harm the protocol;
- Provide clear and comprehensive documentation on any privileged actions within the protocol and the roles authorized to execute them;

Status: Mitigated.

Update from the client: Fixed in commit `world-id-contracts@dd7b7f6`.

7.4 [Medium] `PolygonWorldID` sender and receiver can be set by any address

File(s): `PolygonWorldID.sol`, `StateBridge.sol`

Description: The `PolygonWorldID` contract is responsible for managing the root history of the `WorldID` Merkle root on Polygon PoS. Each new root insertion is called by `StateBridge` (`L1`). The mechanism that is used for sending the state from Ethereum to Polygon PoS consists of two actors: Child and Root. The `PolygonWorldID` is the Child since it's a message receiver, and the `StateBridge` is a Root that sends a state. Both contracts inherit Child and Root functionality from FxBaseChildTunnel and FxBaseRootTunnel, respectively. The issue is that functions for setting Child address in Root contract, and Root address in Child contract are `public`/ `external`. This creates a situation where a transaction for setting the `Child` and the `Root` may be front-run by an adversarial actor. Below we present those functions:

FxBaseChildTunnel.setFxRootTunnel(...)

```
function setFxRootTunnel(address _fxRootTunnel) external virtual {
    require(fxRootTunnel == address(0x0), "FxBaseChildTunnel: ROOT_TUNNEL_ALREADY_SET");
    fxRootTunnel = _fxRootTunnel;
}
```

FxBaseRootTunnel.setFxChildTunnel(...)

```
function setFxChildTunnel(address _fxChildTunnel) public virtual {
    require(fxChildTunnel == address(0x0), "FxBaseRootTunnel: CHILD_TUNNEL_ALREADY_SET");
    fxChildTunnel = _fxChildTunnel;
}
```

Recommendation(s): There are at least two approaches for solving this issue: a) set the Child/Root address in the constructor when deploying contracts. Addresses of deployed contracts can be known upfront; b) overwrite the setting functions and add the `onlyOwner` modifier to prevent front-running the call.

Status: Fixed.

Update from the client: Fixed in commit `world-id-state-bridge@5310dfa`.

7.5 [Low] Absence of `ROOT_HISTORY_EXPIRY` update mechanism in `StateBridge` contract

File(s): `OpWorldID.sol`

Description: The `OpWorldID` contract uses the `onlyOwner` modifier for the functions `receiveRoot(...)` and `setRootHistoryExpiry(...)`, as illustrated below.

```
contract OpWorldID is WorldIDBridge, CrossDomainOwnable3, IOpWorldID {
    ...
    function receiveRoot(...) external virtual onlyOwner {
        _receiveRoot(newRoot, supersedeTimestamp);
    }

   ///////////////////////////////////////////////////////////////
   // @audit Only the StateBridge contract can call this function
   ///////////////////////////////////////////////////////////////
   function setRootHistoryExpiry(...) public virtual override onlyOwner {
        _setRootHistoryExpiry(expiryTime);
    }
}
```

The `receiveRoot(...)` function is called by the `StateBridge` contract (the owner) to forward the latest `root` to the bridged WorldID. In contrast, `setRootHistoryExpiry(...)` updates the expiration time for a `root`. However, the `StateBridge` contract does not have a function for calling `setRootHistoryExpiry(...)` to update the expiration time. To invoke this function, the owner of `StateBridge` must transfer ownership of `OpWorldID` to another contract or to a local Optimism EOA.

Recommendation(s): Add an external function in the `StateBridge` contract that can only be called by the owner to invoke `OpWorldID.setRootHistoryExpiry`

Status: Fixed.

Update from the client: Fixed in commit `world-id-state-bridge@ee74770`.

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

File(s): `WorldIDImpl.sol`

Description: The contract `WorldIDImpl` is derived from OpenZeppelin's `OwnableUpgradeable` contract, which furnishes a single-step ownership transfer. The `transferOwnership(...)` function facilitates the transfer of ownership of the contract in one step. However, if the contract's owner is assigned to an address that is not managed by the Worldcoin team, reclamation of ownership of the contract will not be feasible.

Failing to use such a mechanism can introduce several issues. If the ownership transfer is not properly controlled and executed, it can create security vulnerabilities in the contract. Without proper access control, a malicious actor could gain control of the contract and perform unauthorized actions.

Changing ownership without proper access controls and protocols can lead to a lack of transparency and accountability in the transfer process. This can harm the trust and credibility of the contract and its stakeholders.

Recommendation(s): Consider implementing a two-step process for transferring ownership, such as the propose-accept scheme. Check the Ownable2StepUpgradeable.sol contract from OpenZeppelin.

Status: Fixed.

Update from the client: Fixed in commit `world-id-contracts@c8a10af`.

7.7 [Low] `NonExistentRoot()` may not be executed for nonexistent roots

File(s): `WorldIDBridge.sol`, `WorldIDIdentityManagerImplV1.sol`

Description: The function `checkValidRoot(...)` checks if a given root value is valid. When the `root` is not the latest, the next `if` statement is executed. In case the `root` does not exist, the `rootTimestamp` value is zero, and the function reverts with the error `ExpiredRoot()`, and the condition `if (rootTimestamp == 0)` will not be reached.

```
function checkValidRoot(uint256 root) public view returns (bool isValid) {
    if (root == _latestRoot) {
        return true;
    }

    uint128 rootTimestamp = rootHistory[root];

    if (block.timestamp - rootTimestamp > ROOT_HISTORY_EXPIRY) {
        revert ExpiredRoot();
    }

    ////////////////////////////////////////////////////////////////////
    // @audit The "if" condition below may not be reached when
    // "block.timestamp > ROOT_HISTORY_EXPIRY" and rootTimestamp = 0
    ////////////////////////////////////////////////////////////////////
    if (rootTimestamp == 0) {
        revert NonExistentRoot();
    }
}
```

Recommendation(s): Switch the order of the `if` statements as shown below.

```
-// Expired roots are not valid.
-if (block.timestamp - rootTimestamp > ROOT_HISTORY_EXPIRY) {
-   revert ExpiredRoot();
-}
-
-// And roots do not exist if they don't have an associated timestamp.
-if (rootTimestamp == 0) {
-   revert NonExistentRoot();
-}
+ // And roots do not exist if they don't have an associated timestamp.
+ if (rootTimestamp == 0) {
+     revert NonExistentRoot();
+ }
+ // Expired roots are not valid.
+ if (block.timestamp - rootTimestamp > ROOT_HISTORY_EXPIRY) {
+     revert ExpiredRoot();
+ }
```

Status: Fixed.

Update from the client: Fixed in commits `world-id-contracts@95691df`, `world-id-state-bridge@5a340f6`.

7.8 [Low] `actionId` is hashed two times in `WorldIDAirdrop` contract

File(s): `WorldIDAirdrop.sol`

Description: The `WorldIDAirdrop` constructor requires several parameters, as outlined below.

```
constructor(IWorldIDGroups _worldIdRouter, uint256 _groupId, string memory _actionId,
            ERC20 _token, address _holder, uint256 _airdropAmount) {
  worldIdRouter = _worldIdRouter;
  groupId = _groupId;
  actionId = abi.encodePacked(_actionId).hashToField();
  token = _token;
  holder = _holder;
  airdropAmount = _airdropAmount;
}
```

The constructor accepts the `_actionId` as registered in the developer portal, creates a `keccak256` hash of it, and stores the hash in the `actionId` state variable. When a user claims their reward, they provide the data proving their membership in the group, as demonstrated in the function below.

```
function claim(address receiver, uint256 root, uint256 nullifierHash, uint256[8] calldata proof) public
{
  if (nullifierHashes[nullifierHash]) revert InvalidNullifier();
  worldIdRouter.verifyProof(
      groupId,
      root,
      abi.encodePacked(receiver).hashToField(), // The signal of the proof
      nullifierHash,
      ///////////////////////////////////////////////////////////////
      // @audit the "actionId""" is already hashed
      //        in the constructor
      ///////////////////////////////////////////////////////////////
    abi.encodePacked(actionId).hashToField(), // The external nullifier hash
      proof
  );
...
}
```

However, the `claim(...)` function hashes the already hashed `actionId` once more. If the proof is not generated for the double-hashed `actionId`, the `worldIdRouter.verifyProof` will revert.

Recommendation(s): Remove the double hashing.

```
function claim(address receiver, uint256 root, uint256 nullifierHash, uint256[8] calldata proof) public
{
  if (nullifierHashes[nullifierHash]) revert InvalidNullifier();
  worldIdRouter.verifyProof(
      groupId,
      root,
      abi.encodePacked(receiver).hashToField(), // The signal of the proof
      nullifierHash,
      ////////////////////////////////////////
      // @audit the actionId is already hashed
      //        in the constructor
      ///////////////////////////////////////
-         abi.encodePacked(actionId).hashToField(), // The external nullifier hash
+         actionId, // The external nullifier hash
      proof
  );
...
}
```

Status: Fixed.

Update from the client: Fixed in commit `world-id-example-airdrop@f7b0cee`.

7.9 [Undetermined] `ActionID` may not be unique across applications

Description: When verifying a claim through `WorldIDGroups`, a user must provide six elements:

- `groupId`: A group to which the user belongs;
- `root`: The root of the Merkle tree containing all identities in the group;
- `signalHash`: The hash of a message the user commits to;
- `nullifierHash`: An identifier for the proof, derived from the user's identity and `ActionId`;
- `externalNullifierHash`: The `ActionId` represents the user's desired action;
- `proof`: Eight values used to validate the provided arguments;

Provided all the values are correct, the caller proves that:

- The user is part of the group identified by `groupId`;
- The user wants to perform the action represented by `ActionId/ExternalNullifierHash`;
- The user commits to the information represented by `signalHash` for their action;

Although `WorldIDGroup` does not prevent reusing proofs, applications built on top of it can do so using the `nullifierHash`. Allowing a specific `nullifierHash` to be used only once means each user can execute the action identified by `externalNullifierHash` only once.

However, since this check is performed in each application, proofs can still be reused across different applications. This can be mitigated by using unique `ActionIDs` for each application. The `WorldCoin` team recommends building `ActionIDs` using the application address as a prefix, but there are no restrictions preventing a malicious user from using the same `ActionIDs` as other applications.

An attacker could create a phishing application and incentivize users to submit proofs to it, then reuse these proofs in other applications using `WorldID`. The severity and likelihood of this attack vary across applications.

Recommendation(s): In order to mitigate this issue, we recommend doing the following:

- Encourage applications using `WorldID` to follow a well-defined standard for creating `ActionIds`. The `ActionIds` should include identifiers unique to the application and easily verifiable by users, such as the address of the contract;
- Develop tools that allow users to inspect a human-readable version of the `ActionIds`. For example, in addition to the `ActionId`, display the values used to generate it, which could include application identifiers such as the address;
- Encourage users to carefully review the `ActionIds` they are signaling when submitting proofs;

Status: Acknowledged

Update from the client: Working on documentation to explain how this will be/is handled.

7.10 [Info] Double initialization of the tree depth

File(s): PolygonWorldID.sol

Description: The constructor of the PolygonWorldID contract checks and sets the tree depth, as we can see in the code below.

```
contract PolygonWorldID is WorldIDBridge, FxBaseChildTunnel, Ownable {

    constructor(uint8 _treeDepth, address _fxChild)
        WorldIDBridge(_treeDepth)
        FxBaseChildTunnel(_fxChild)
    {
        if (!SemaphoreTreeDepthValidator.validate(_treeDepth)) {
            revert UnsupportedTreeDepth(_treeDepth);
        }
        ///////////////////////////////////
        // @audit tree depth is set here //
        ///////////////////////////////////
        treeDepth = _treeDepth;
    }
```

However, the identical process occurs within the WorldIDBridge contract, from which the PolygonWorldID contract inherits. This results in a superfluous assignment of the tree depth.

```
abstract contract WorldIDBridge is IWorldID {
    ...
    constructor(uint8 _treeDepth) {
        if (!SemaphoreTreeDepthValidator.validate(_treeDepth)) {
            revert UnsupportedTreeDepth(_treeDepth);
        }
        ////////////////////////////////////////
        // @audit tree depth is also set here //
        ////////////////////////////////////////
        treeDepth = _treeDepth;
    }
}
```

Recommendation(s): Avoid redundant operations.

Status: Fixed.

Update from the client: Fixed in commit world-id-state-bridge@25e6d85.

7.11 [Info] Unnecessary storage writes

File(s): WorldIDRouterImplV1.sol

Description: The WorldIDRouter contract utilizes an array as a routingTable, which stores all registered groups and identifies them by their index in the array. The array is initialized with a length of DEFAULT_ROUTING_TABLE_SIZE. When a new group is added, and the current number of elements in the routingTable equals its size, a new array is created in memory with a length equal to the current size plus DEFAULT_ROUTING_TABLE_GROWTH. The newly created array is then copied element by element to the storage.

```
function insertNewTableEntry(uint256 groupId, address targetAddress)
    internal
    onlyProxy
    onlyInitialized
{
    while (groupId >= routingTable.length) {
        uint256 existingTableLength = routingTable.length;
        address[] memory newRoutingTable =
            new address[](existingTableLength + DEFAULT_ROUTING_TABLE_GROWTH);

        for (uint256 i = 0; i < existingTableLength; ++i) {
            newRoutingTable[i] = routingTable[i];
        }
        // @audit - This operation overwrites storage values, causing
        //          unnecessary gas consumption.
        routingTable = newRoutingTable;
    }

    routingTable[groupId] = targetAddress;
    _groupCount++;
}
```

This operation overwrites existing storage values with the same values, leading to unnecessary gas consumption.

Recommendation(s): Consider removing reallocation logic and using the push method for storage arrays instead.

```
- /// The default size of the internal routing table.
- uint256 internal constant DEFAULT_ROUTING_TABLE_SIZE = 10;

- /// How much the routing table grows when it runs out of space.
- uint256 internal constant DEFAULT_ROUTING_TABLE_GROWTH = 5;

/// The null address.
address internal constant NULL_ADDRESS = address(0x0);

/// The routing table is used to dispatch from groups to addresses.
address[] internal routingTable;

- /// The number of groups currently set in the routing table.
- uint256 internal _groupCount;
```

```
function initialize(address initialGroupIdentityManager) public reinitializer(1) {
    // Initialize the sub-contracts.
    __delegateInit();

    // Now we can perform our own internal initialisation.
-   routingTable = new address[](DEFAULT_ROUTING_TABLE_SIZE);
-   routingTable[0] = initialGroupIdentityManager;
-   _groupCount = 1;
+   routingTable.push(initialGroupIdentityManager);

    // Mark the contract as initialized.
    __setInitialized();
}
```

```
function groupCount() public view onlyProxy onlyInitialized returns (uint256 count) {
-   return _groupCount;
+   return routingTable.length;
}
```

```
function insertNewTableEntry(uint256 groupId, address targetAddress)
    internal
    onlyProxy
    onlyInitialized
{
-   while (groupId >= routingTable.length) {
-       uint256 existingTableLength = routingTable.length;
-       address[] memory newRoutingTable =
-           new address[](existingTableLength + DEFAULT_ROUTING_TABLE_GROWTH);
-
-       for (uint256 i = 0; i < existingTableLength; ++i) {
-           newRoutingTable[i] = routingTable[i];
-       }
-
-       routingTable = newRoutingTable;
-   }
-
-   routingTable[groupId] = targetAddress;
-   _groupCount++;
+   routingTable.push(targetAddress);
}
```

```
function nextGroupId() public view onlyProxy onlyInitialized returns (uint256 count) {
-   return _groupCount;
+   return routingTable.length;
}
```

Status: Fixed.

Update from the client: Fixed in commits world-id-contracts@95691df, world-id-contracts@38a5d1c.

7.12 [Best Practice] Lack of events for relevant operations

File(s): WorldIDIdentityManagerImplV1.sol, WorldIDRouterImplV1.sol, WorldIDBridge.sol, WorldIDAirdrop.sol

Description: Several relevant operations in the contracts do not emit events, making it difficult to monitor and review the contracts' behavior once deployed.

Operations that would benefit from emitting events include:

```
- WorldIDIdentityManagerImplV1.initialize(...)
- WorldIDIdentityManagerImplV1.registerIdentities(...)
- WorldIDIdentityManagerImplV1.updateIdentities(...)
- WorldIDIdentityManagerImplV1.setStateBridge(...)
- WorldIDIdentityManagerImplV1.setSemaphoreVerifier(...)
- WorldIDIdentityManagerImplV1.setRootHistoryExpiry(...)
- WorldIDRouterImplV1.initialize(...)
- WorldIDRouterImplV1.addGroup(...)
- WorldIDRouterImplV1.updateGroup(...)
- WorldIDBridge._setRootHistoryExpiry(...)
- WorldIDAirdrop.claim(...)
```

Emitting events for the relevant operations will enable users and blockchain monitoring systems to easily detect suspicious behaviors and ensure the correct functioning of the contracts.

Recommendation(s): Add events for all relevant operations to facilitate contract monitoring and detect suspicious behavior more effectively.

Status: Fixed.

Update from the client: Fixed in commits world-id-contracts@5f0f56c, world-id-state-bridge@aab1ea1, world-id-example-airdrop@f7d60fe, world-id-contracts@d29d6a9.

7.13 [Best Practice] Low-level calls may cause errors

File(s): WorldIDRouterImplV1.sol

Description:

The `verifyProof(...)` function, present within the `WorldIDRouter` contract, retrieves the appropriate `IdentityManager` for the indicated group and invokes the `verifyProof(...)` function of that `IdentityManager`. This call to the `IdentityManager` is accomplished by employing the following code.

```
bytes memory callData = abi.encodeCall(
    IWorldID.verifyProof, (root, signalHash, nullifierHash, externalNullifierHash, proof)
);

(bool success,) = identityManager.call(callData);
if (!success) {
    revert FailedToVerifyProof();
}
```

As demonstrated in the code excerpt provided, a low-level `call` instruction is utilized. Utilizing low-level calls is not advisable as they do not ordinarily implement security precautions. For instance, in this particular scenario, if the address pointed to by `identityManager` lacks any code, this function will succeed when it should actually revert.

Recommendation(s): Avoid using low-level calls. Use `IWorldID` as the type of `identityManager`.

Status: Fixed.

Update from the client: Fixed in commit world-id-contracts@95691df.

7.14 [Best Practice] Non-private functions are not virtual

File(s): WorldIDRouterImplV1.sol

Description: The optimal practices for implementing contracts are defined at the commencement of `WorldIDRouterImplV1` contract. Nevertheless, the advice to make all non-private functions virtual has not been strictly adhered to, as there exist non-private functions that lack virtual property. This might result in an inability to update these functions if a new implementation were to inherit from the present one.

Recommendation(s): Define non-private functions as virtual in implementation contracts.

Status: Fixed.

Update from the client: Fixed in commits world-id-contracts@95691df, world-id-contracts@5034fec.

7.15 [Best Practice] Presence of unused variables

File(s): StateBridge.sol, WorldIDBridge.sol

Description: The presence of unused variables is generally considered a practice to be avoided, as they may lead to potential issues such as:

- Increased computational costs (i.e., unnecessary gas consumption);
- Reduced code readability; and;
- The possibility of bugs (e.g., when an explicit return statement is forgotten and return values are never assigned);

The state variable `worldID` in the `StateBridge` contract is only set in the constructor and is not used throughout the contract.

```

contract StateBridge is FxBaseRootTunnel, Ownable {
...
    // @audit unused state variable
    IWorldIDIdentityManager internal worldID;
...
}

```

Another example of an unused variable is present in the `WorldIDBridge` contract, declared in the function `checkValidRoot(...)`. The function contains the local variable `isValid`, which is never assigned a value, so it assumes its initialization value. When the `if` condition is not satisfied, `false` will always be returned without any warning of the forgotten statement `return`.

```
abstract contract WorldIDBridge is IWorldID  {
...
    function checkValidRoot(uint256 root) public view returns (bool isValid) {
        // @audit a value is never assigned to isValid
        if (root == _latestRoot) {
            return true;
        }

        uint128 rootTimestamp = rootHistory[root];

        // Expired roots are not valid.
        if (block.timestamp - rootTimestamp > ROOT_HISTORY_EXPIRY) {
            revert ExpiredRoot();
        }

        // And roots do not exist if they don't have an associated timestamp.
        if (rootTimestamp == 0) {
            revert NonExistentRoot();
        }
    }
...
}
```

List of unused local variables in `return` statements:

```
- WorldIDRouterImplV1
  - function routeFor(...)

- WorldIDBridge
   - function checkValidRoot
   - function latestRoot()
   - function rootHistoryExpiry()
   - function getTreeDepth()

- WorldIDIdentityManagerImplV1
   - function queryRoot(...)
   - function latestRoot() stateBridge()
   - function isInputInReducedForm(...)
   - function reduceInputElementInSnarkScalarField(...)
   - function getRegisterIdentitiesVerifierLookupTableAddress()
   - function getIdentityUpdateVerifierLookupTableAddress()
   - function getSemaphoreVerifierAddress()
   - function getRootHistoryExpiry()
   - function getTreeDepth()
```

Unused state variable:

```
StateBridge
  - IWorldIDIdentityManager internal worldID
```

Recommendation(s): We recommend removing all unused variables from the codebase to enhance code quality and minimize the risk of unexpected errors or inefficiencies.

Status: Fixed.

Update from the client: Fixed in commits `world-id-state-bridge@93fc506`, world-id-state-bridge@1f8366c, world-id-contracts@275e2aa.

8 Documentation Evaluation

Software documentation refers to written or visual information describing software's functionality, architecture, design, and implementation. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

To assist the audit process, the World ID team provided Unit Tests, Worldcoin docs on the Github repository, and extensive documentation available here. The documentation embraces detailed specifications for the contracts under auditing, e.g., how World ID proofs are verified on-chain and how they use Zero-Knowledge proofs. Notably, their code is rich in inline comments, significantly contributing to understanding the implementation and functionality.

In addition to the official documentation, we also rely on documentation for:

- Semaphore;
- Bridge system used for sending messages to Polygon;
- Bridge system used for sending messages to Optimism.

The documentation provided by the WorldCoin team and the additional sources we utilized enabled us to thoroughly cover the terms used in the source code, core business logic, and function flow. Reviewing the entire documentation suite, we comprehensively understood how the contracts should function. The provided documentation is well-written and structured, making it an excellent resource for developers and auditors. The codebase contains an adequate number of inline comments and well-named functions and variables, which greatly assisted the audit team in understanding the function flow and identifying any potential issues.

9 Test Suite Evaluation

9.1 Contracts Compilation Output

9.1.1 World ID

```
> make build
forge build
[] Compiling...
[] Compiling 75 files with 0.8.19
[] Solc 0.8.19 finished in 42.89s
Compiler run successful
```

9.1.2 State Bridge

```
> make build
forge build
[] Compiling...
[] Compiling 126 files with 0.8.15
[] Solc 0.8.15 finished in 15.36s
Compiler run successful (with warnings)
warning[6321]: Warning: Unnamed return variable can remain unassigned. Add an explicit return with value to all
↪  non-reverting code paths or name the variable.
  --> lib/contracts/contracts/lib/MerklePatriciaProof.sol:20:30:
   |
20 |     ) internal pure returns (bool) {
   |                              ^^^^



warning[5667]: Warning: Unused function parameter. Remove or comment out the variable name to silence this warning.
  --> src/mock/WorldIDIdentityManagerMock.sol:25:29:
   |
25 |     function checkValidRoot(uint256 root) public view returns (bool) {
   |                             ^^^^^^^^^^^^



warning[2018]: Warning: Function state mutability can be restricted to pure
  --> src/mock/WorldIDIdentityManagerMock.sol:25:5:
   |
25 |     function checkValidRoot(uint256 root) public view returns (bool) {
   |     ^ (Relevant source part starts here and spans across multiple lines).
```

9.1.3 Airdrop Example

```
> make build
forge build
[] Compiling...
[] Compiling 14 files with 0.8.19
[] Solc 0.8.19 finished in 2.06s
Compiler run successful
```

9.2 Tests Output

9.2.1 World ID

```
> make test
FOUNDRY_PROFILE=debug forge test
[] Compiling...
[] Compiling 27 files with 0.8.19
[] Solc 0.8.19 finished in 44.35s
Compiler run successful

Running 1 test for src/test/verifier-lookup-table/VerifierLookupTableConstruction.t.sol:BatchLookupTableConstruction
[PASS] testCanConstructLookupTable() (gas: 268426)
Test result: ok. 1 passed; 0 failed; finished in 1.74ms

Running 2 tests for src/test/router/WorldIDRouterDataQuery.t.sol:WorldIDRouterDataQuery
[PASS] testCanGetGroupCount() (gas: 24168)
[PASS] testCannotGetGroupCountUnlessViaProxy() (gas: 8555)
Test result: ok. 2 passed; 0 failed; finished in 6.90ms

Running 5 tests for
↪ src/test/verifier-lookup-table/VerifierLookupTableOwnershipManagement.t.sol:BatchLookupTableOwnershipManagement
[PASS] testCannotRenounceOwnershipIfNotOwner(address) (runs: 256, : 15830, ~: 15830)
[PASS] testCannotTransferOwnerIfNotOwner(address,address) (runs: 256, : 16119, ~: 16119)
[PASS] testHasOwner() (gas: 9869)
[PASS] testRenounceOwnership() (gas: 10530)
[PASS] testTransferOwner(address) (runs: 256, : 22699, ~: 22699)
Test result: ok. 5 passed; 0 failed; finished in 67.67ms

Running 9 tests for src/test/verifier-lookup-table/VerifierLookupTableQuery.t.sol.sol:VerifierLookupTableQuery
[PASS] testCanAddVerifierWithValidBatchSize(uint256,address) (runs: 256, : 39450, ~: 39450)
[PASS] testCanDisableVerifier(uint256) (runs: 256, : 28202, ~: 28225)
[PASS] testCanGetVerifierForExtantBatchSize() (gas: 12307)
[PASS] testCanUpdateVerifierWithValidBatchSize(address) (runs: 256, : 23916, ~: 23916)
[PASS] testCannotAddVerifierForBatchSizeThatAlreadyExists(uint256) (runs: 256, : 39342, ~: 39342)
[PASS] testCannotAddVerifierUnlessOwner(address) (runs: 256, : 18123, ~: 18123)
[PASS] testCannotDisableVerifierUnlessOwner(address) (runs: 256, : 18089, ~: 18089)
[PASS] testCannotGetVerifierForMissingBatchSize(uint256) (runs: 256, : 13596, ~: 13596)
[PASS] testCannotUpdateVerifierIfNotOwner(address) (runs: 256, : 20353, ~: 20353)
Test result: ok. 9 passed; 0 failed; finished in 225.43ms

Running 5 tests for src/test/router/WorldIDRouterOwnershipManagement.t.sol:WorldIDRouterOwnershipManagement
[PASS] testCannotRenounceOwnershipIfNotOwner(address) (runs: 256, : 48701, ~: 48711)
[PASS] testCannotTransferOwnerIfNotOwner(address,address) (runs: 256, : 49135, ~: 49135)
[PASS] testHasOwner() (gas: 21821)
[PASS] testRenounceOwnership() (gas: 25248)
[PASS] testTransferOwner(address) (runs: 256, : 37763, ~: 37763)
Test result: ok. 5 passed; 0 failed; finished in 385.85ms

Running 15 tests for src/test/identity-manager/WorldIDIdentityManagerUninit.t.sol:WorldIDIdentityManagerUninit
[PASS] testShouldNotCallCalculateIdentityRegistrationInputHash() (gas: 3167241)
[PASS] testShouldNotCallCheckValidRootWhileUninit() (gas: 3157481)
[PASS] testShouldNotCallGetIdentityUpdateVerifierLookupTableAddressWhileUninit() (gas: 3157309)
[PASS] testShouldNotCallGetRootHistoryExpiryWhileUninit() (gas: 3157199)
[PASS] testShouldNotCallGetSemaphoreVerifierAddressWhileUninit() (gas: 3157285)
[PASS] testShouldNotCallIsInputInReducedFormWhileUninit() (gas: 3157414)
[PASS] testShouldNotCallLatestRootWhileUninit() (gas: 3157308)
[PASS] testShouldNotCallQueryRootWhileUninit() (gas: 3157583)
[PASS] testShouldNotCallRegisterIdentitiesWhileUninit() (gas: 3185451)
[PASS] testShouldNotCallSetIdentityUpdateVerifierLookupTableWhileUninit() (gas: 3951369)
[PASS] testShouldNotCallSetRegisterIdentitiesVerifierLookupTableWhileUninit() (gas: 3951414)
[PASS] testShouldNotCallSetRootHistoryExpiryWhileUninit() (gas: 3157465)
[PASS] testShouldNotCallSetSemaphoreVerifierWhileUninit() (gas: 9349980)
[PASS] testShouldNotCallUpdateIdentitiesWhileUninit(uint128[],uint128[8]) (runs: 256, : 3359014, ~: 3367924)
[PASS] testShouldNotCallgetRegisterIdentitiesVerifierLookupTableAddressWhileUninit() (gas: 3157309)
Test result: ok. 15 passed; 0 failed; finished in 1.18s
```

```
Running 7 tests for src/test/identity-manager/WorldIDIdentityManagerCalculation.t.sol:WorldIDIdentityManagerCalculation
[PASS] testCalculateIdentityRegistrationInputHashFromParametersOnKnownInput() (gas: 32679)
[PASS] testCanCalculateIdentityUpdateInputHash(uint256,uint256,uint32,uint32,uint256,uint256,uint256,uint256) (runs:
↪ 256, : 27471, ~: 27471)

[PASS] testCanCheckValueIsInReducedForm(uint256) (runs: 256, : 25369, ~: 25369)
[PASS] testCanCheckValueIsNotInReducedForm(uint256) (runs: 256, : 25398, ~: 25398)
[PASS] testCannotCalculateIdentityRegistrationInputHashIfNotViaProxy() (gas: 17895)
[PASS] testCannotCalculateIdentityUpdateHashIfNotViaProxy(uint256,uint256,uint32[],uint256[],uint256[]) (runs: 256, :
↪ 79263, ~: 81179)
[PASS] testCannotCheckValidIsInReducedFormIfNotViaProxy() (gas: 8702)
Test result: ok. 7 passed; 0 failed; finished in 1.22s

Running 2 tests for
↪ src/test/identity-manager/WorldIDIdentityManagerConstruction.t.sol:WorldIDIdentityManagerConstruction
[PASS] testCanConstructIdentityManagerWithDelegate() (gas: 3345748)
[PASS] testCanConstructIdentityManagerWithNoDelegate() (gas: 97722)
Test result: ok. 2 passed; 0 failed; finished in 10.11ms

Running 2 tests for src/test/router/WorldIDRouterConstruction.t.sol:WorldIDRouterConstruction
[PASS] testCanConstructRouterWithDelegate(address) (runs: 256, : 1687484, ~: 1687484)
[PASS] testCanConstructRouterWithNoDelegate() (gas: 97679)
Test result: ok. 2 passed; 0 failed; finished in 158.51ms

Running 4 tests for src/test/identity-manager/WorldIDIdentityManagerUpgrade.t.sol:WorldIDIdentityManagerUpdate
[PASS] testCanUpgradeImplementationWithCall() (gas: 3138170)
[PASS] testCanUpgradeImplementationWithoutCall() (gas: 3124384)
[PASS] testCannotUpgradeUnlessManager(address) (runs: 256, : 3149466, ~: 3149466)
[PASS] testCannotUpgradeWithoutProxy() (gas: 3125858)
Test result: ok. 4 passed; 0 failed; finished in 463.88ms

Running 4 tests for
↪ src/test/identity-manager/WorldIDIdentityManagerInitialization.t.sol:WorldIDIdentityManagerInitialization
[PASS] testCannotInitializeTheDelegate() (gas: 3056958)
[PASS] testCannotPassUnsupportedTreeDepth() (gas: 3176393)
[PASS] testInitialisation() (gas: 3351225)
[PASS] testInitializationOnlyOnce() (gas: 62063)
Test result: ok. 4 passed; 0 failed; finished in 23.98ms

Running 21 tests for
↪ src/test/identity-manager/WorldIDIdentityManagerGettersSetters.t.sol:WorldIDIdentityManagerGettersSetters
[PASS] testCanGetIdentityUpdateVerifierLookupTableAddress() (gas: 26479)
[PASS] testCanGetRegisterIdentitiesVerifierLookupTableAddress() (gas: 26488)
[PASS] testCanGetRootHistoryExpiry() (gas: 24286)
[PASS] testCanGetSemaphoreVerifierAddress() (gas: 15236)
[PASS] testCanSetIdentityUpdateVerifierLookupTable() (gas: 825574)
[PASS] testCanSetRegisterIdentitiesVerifierLookupTable() (gas: 825554)
[PASS] testCanSetRootHistoryExpiry(uint256) (runs: 256, : 34474, ~: 34474)
[PASS] testCanSetSemaphoreVerifier() (gas: 6225791)
[PASS] testCannotGetIdentityUpdateVerifierLookupTableAddressUnlessViaProxy() (gas: 8709)
[PASS] testCannotGetRegisterIdentitiesVerifierLookupTableAddressUnlessViaProxy() (gas: 8707)
[PASS] testCannotGetRootHistoryExpiryUnlessViaProxy() (gas: 8635)
[PASS] testCannotGetSemaphoreVerifierAddressUnlessViaProxy() (gas: 8664)
[PASS] testCannotSetIdentityUpdateVerifierLookupTableUnlessOwner(address) (runs: 256, : 841169, ~: 841169)
[PASS] testCannotSetIdentityUpdateVerifierLookupTableUnlessViaProxy() (gas: 802699)
[PASS] testCannotSetRegisterIdentitiesVerifierLookupTableUnlessOwner(address) (runs: 256, : 841147, ~: 841147)
[PASS] testCannotSetRegisterIdentitiesVerifierLookupTableUnlessViaProxy() (gas: 802702)
[PASS] testCannotSetRootHistoryExpiryToZero() (gas: 43650)
[PASS] testCannotSetRootHistoryExpiryUnlessOwner(address) (runs: 256, : 6241419, ~: 6241419)
[PASS] testCannotSetRootHistoryExpiryUnlessViaProxy(uint256) (runs: 256, : 9145, ~: 9145)
[PASS] testCannotSetSemaphoreVerifierAddressUnlessViaProxy() (gas: 6202971)
[PASS] testCannotSetSemaphoreVerifierUnlessOwner(address) (runs: 256, : 6241375, ~: 6241375)
Test result: ok. 21 passed; 0 failed; finished in 1.69s

Running 5 tests for
↪ src/test/identity-manager/WorldIDIdentityManagerOwnershipManagement.t.sol:WorldIDIdentityManagerOwnershipManagement
[PASS] testCannotRenounceOwnershipIfNotOwner(address) (runs: 256, : 48782, ~: 48782)
[PASS] testCannotTransferOwnerIfNotOwner(address,address) (runs: 256, : 49177, ~: 49177)
[PASS] testHasOwner() (gas: 21854)
[PASS] testRenounceOwnership() (gas: 25314)
[PASS] testTransferOwner(address) (runs: 256, : 37839, ~: 37839)
Test result: ok. 5 passed; 0 failed; finished in 411.44ms
```

```
Running 5 tests for src/test/router/WorldIDRouterUninit.t.sol:WorldIDRouterUninit
[PASS] testCannotAddGroupWhileUninit(uint256,address) (runs: 256, : 1538875, ~: 1538875)
[PASS] testCannotDisableGroupWhileUninit(uint256) (runs: 256, : 1538593, ~: 1538593)
[PASS] testCannotGetGroupCountWhileUninit() (gas: 1538294)

[PASS] testCannotGetRouteForWhileUninit(uint256) (runs: 256, : 1538549, ~: 1538549)
[PASS] testCannotUpdateGroupWhileUninit(uint256,address) (runs: 256, : 1538966, ~: 1538966)
Test result: ok. 5 passed; 0 failed; finished in 705.28ms

Running 7 tests for src/test/identity-manager/WorldIDIdentityManagerStateBridge.t.sol:WorldIDIdentityManagerStateBridge
[PASS] testCanDisableStateBridgeFunctionality() (gas: 5406712)
[PASS] testCanEnableStateBridgeIfDisabled() (gas: 3370092)
[PASS] testCanUpgradestateBridge(address) (runs: 256, : 24107, ~: 24107)
[PASS] testCannotDisableStateBridgeIfAlreadyDisabled() (gas: 23999)
[PASS] testCannotEnableStateBridgeIfAlreadyEnabled() (gas: 18776)
[PASS] testCannotUpdateStateBridgeAsNonOwner(address) (runs: 256, : 47185, ~: 47185)
[PASS] testCannotUpgradeStateBridgeToZeroAddress() (gas: 17008)
Test result: ok. 7 passed; 0 failed; finished in 201.28ms

Running 4 tests for src/test/router/WorldIDRouterUpgrade.t.sol:WorldIDRouterUpgrade
[PASS] testCanUpgradeImplementationWithCall() (gas: 1535565)
[PASS] testCanUpgradeImplementationWithoutCall() (gas: 1504686)
[PASS] testCannotUpgradeUnlessManager(address) (runs: 256, : 1529864, ~: 1529864)
[PASS] testCannotUpgradeWithoutProxy(address) (runs: 256, : 1493187, ~: 1493187)
Test result: ok. 4 passed; 0 failed; finished in 229.40ms

Running 2 tests for
→ src/test/identity-manager/WorldIDIdentityManagerSemaphoreVerification.t.sol:WorldIDIdentityManagerSemaphoreValidation
[PASS] testProofVerificationWithCorrectInputs(uint8,uint256,uint256,uint256,uint256[8]) (runs: 256, : 3484575, ~:
→ 3484575)
[PASS] testProofVerificationWithInorrectProof(uint8,uint256,uint256,uint256,uint256[8]) (runs: 256, : 3468459, ~:
→ 3468459)
Test result: ok. 2 passed; 0 failed; finished in 3.15s

Running 3 tests for src/test/router/WorldIDRouterStateBridge.t.sol:WorldIDRouterStateBridge
[PASS] testCanAddStateBridgeAsGroup(uint8,address) (runs: 256, : 2879543, ~: 1495851)
[PASS] testCanProxyVerifyProofForStateBridge(uint8,uint256,uint256,uint256,uint256,uint256[8]) (runs: 256, : 2381268,
→ ~: 796816)
[PASS] testCanUpdateStateBridgeAsGroup(uint8) (runs: 256, : 2610789, ~: 1174729)
Test result: ok. 3 passed; 0 failed; finished in 14.12s

Running 17 tests for src/test/router/WorldIDRouterRouting.t.sol:WorldIDRouterRouting
[PASS] testCanAddGroup(address) (runs: 256, : 64062, ~: 64062)
[PASS] testCanDisableGroup(uint8,address) (runs: 256, : 4615273, ~: 2253427)
[PASS] testCanGetRouteForValidGroup(uint8,address,address) (runs: 256, : 2239281, ~: 607870)
[PASS] testCanUpdateGroup(uint8,address) (runs: 256, : 2581350, ~: 569576)
[PASS] testCannotAddDuplicateGroup(address) (runs: 256, : 28609, ~: 28609)
[PASS] testCannotAddGroupUnlessNumbersSequential(uint256,address) (runs: 256, : 32161, ~: 32161)
[PASS] testCannotAddGroupUnlessOwner(address) (runs: 256, : 51521, ~: 51521)
[PASS] testCannotAddGroupUnlessViaProxy(address) (runs: 256, : 8906, ~: 8906)
[PASS] testCannotDisableGroupUnlessOwner(address) (runs: 256, : 49227, ~: 49227)
[PASS] testCannotDisableGroupUnlessViaProxy(uint256) (runs: 256, : 8863, ~: 8863)
[PASS] testCannotUpdateGroupUnlessOwner(address) (runs: 256, : 51517, ~: 51517)
[PASS] testCannotUpdateGroupUnlessViaProxy(uint256,address) (runs: 256, : 9079, ~: 9079)
[PASS] testShouldRevertOnDisabledGroup(uint8) (runs: 256, : 2673149, ~: 1069852)
[PASS] testShouldRevertOnDisablingNonexistentGroup(uint256) (runs: 256, : 31838, ~: 31838)
[PASS] testShouldRevertOnRouteRequestForMissingGroup(uint256) (runs: 256, : 29052, ~: 29052)
[PASS] testShouldRevertOnUpdatingNonexistentGroup(uint256,address) (runs: 256, : 31572, ~: 31572)
[PASS] testcannotGetRouteUnlessViaProxy(uint256) (runs: 256, : 8796, ~: 8796)
Test result: ok. 17 passed; 0 failed; finished in 13.75s

Running 8 tests for src/test/identity-manager/WorldIDIdentityManagerDataQuery.t.sol:WorldIDIdentityManagerDataQuery
[PASS] testCanGetLatestRoot(uint256) (runs: 256, : 3377890, ~: 3378746)
[PASS] testCanGetTreeDepth(uint8) (runs: 256, : 3381665, ~: 3381665)
[PASS] testCannotGetLatestRootIfNotViaProxy() (gas: 8720)
[PASS] testCannotQueryRootIfNotViaProxy() (gas: 11186)
```

```
[PASS] testQueryCurrentRoot(uint128) (runs: 256, : 3396215, ~: 3397071)
[PASS] testQueryExpiredRoot(uint128[8],uint32,uint128,uint128,uint128[]) (runs: 256, : 4461181, ~: 4452379)
[PASS] testQueryInvalidRoot(uint256) (runs: 256, : 52986, ~: 52986)
[PASS] testQueryOlderRoot(uint128[8],uint32,uint128,uint128,uint128[]) (runs: 256, : 4442717, ~: 4423219)
Test result: ok. 8 passed; 0 failed; finished in 14.81s

Running 10 tests for
↪ src/test/identity-manager/WorldIDIdentityManagerIdentityUpdate.t.sol:WorldIDIdentityManagerIdentityUpdate
[PASS] testCannotRegisterIdentitiesWithOutdatedRoot(uint256,uint256,uint128[],uint128[8]) (runs: 256, : 4377288, ~:
↪ 4378799)
[PASS] testCannotRegisterIdentitiesWithUnreducedIdentities(uint128,uint256,uint128,uint128[],uint128[8],bool) (runs:
↪ 256, : 4404991, ~: 4402297)
[PASS] testCannotUpdateIdentitiesAsNonManager(address,uint128[],uint128[8]) (runs: 256, : 245188, ~: 237648)
[PASS] testCannotUpdateIdentitiesIfNotViaProxy(uint128[],uint128[8]) (runs: 256, : 165693, ~: 153891)
[PASS] testCannotUpdateIdentitiesWithIncorrectInputs(uint128[8],uint128,uint128,uint128[]) (runs: 256, : 4536744, ~:
↪ 4524380)
[PASS] testCannotUpdateIdentitiesWithInvalidBatchSize(uint128[8],uint128,uint128,uint128[]) (runs: 256, : 4544864, ~:
↪ 4540009)
[PASS] testCannotUpdateIdentitiesWithUnreducedPostRoot(uint128,uint128[],uint128[8]) (runs: 256, : 239038, ~: 235796)
[PASS] testCannotUpdateIdentitiesWithUnreducedPreRoot(uint128,uint128[],uint128[8]) (runs: 256, : 227578, ~: 217498)
[PASS] testUpdateIdentitiesSelectsCorrectVerifier(uint128[8],uint128,uint128,uint128[]) (runs: 256, : 5022942, ~:
↪ 4999556)
[PASS] testUpdateIdentitiesWithCorrectInputs(uint128[8],uint128,uint128,uint128[]) (runs: 256, : 4542749, ~: 4520284)
Test result: ok. 10 passed; 0 failed; finished in 14.81s

Running 17 tests for
↪ src/test/identity-manager/WorldIDIdentityManagerIdentityRegistration.t.sol:WorldIDIdentityManagerIdentityRegistration
[PASS] testCannotRegisterIdentitiesAsNonManager(address) (runs: 256, : 75237, ~: 75237)
[PASS] testCannotRegisterIdentitiesIfIdentitiesIncorrect(uint256) (runs: 256, : 5361425, ~: 5361425)
[PASS] testCannotRegisterIdentitiesIfNotViaProxy() (gas: 41017)
[PASS] testCannotRegisterIdentitiesIfPostRootIncorrect(uint256) (runs: 256, : 5337024, ~: 5337024)
[PASS] testCannotRegisterIdentitiesIfStartIndexIncorrect(uint32) (runs: 256, : 5337128, ~: 5337128)
[PASS] testCannotRegisterIdentitiesWithIncorrectInputs(uint128[8],uint32,uint128,uint128,uint128[]) (runs: 256, :
↪ 4400789, ~: 4390435)
[PASS] testCannotRegisterIdentitiesWithInvalidBatchSize(uint128[8],uint32,uint128,uint128,uint128[]) (runs: 256, :
↪ 4397390, ~: 4404074)
[PASS] testCannotRegisterIdentitiesWithInvalidIdentities(uint8,uint8) (runs: 256, : 150921, ~: 161264)
[PASS] testCannotRegisterIdentitiesWithOutdatedRoot(uint256,uint256) (runs: 256, : 3419979, ~: 3420835)
[PASS] testCannotRegisterIdentitiesWithUnreducedIdentities(uint128) (runs: 256, : 85385, ~: 85385)
[PASS] testCannotRegisterIdentitiesWithUnreducedPostRoot(uint128) (runs: 256, : 68134, ~: 68134)
[PASS] testCannotRegisterIdentitiesWithUnreducedPreRoot(uint128) (runs: 256, : 63503, ~: 63503)
[PASS] testCannotRegisterIdentitiesWithUnreducedStartIndex(uint256) (runs: 256, : 41564, ~: 41564)
[PASS] testRegisterIdentitiesSelectsCorrectVerifier(uint128[8],uint32,uint128,uint128,uint128[]) (runs: 256, : 4789172,
↪ ~: 4774604)
[PASS] testRegisterIdentitiesWithCorrectInputs(uint128[8],uint32,uint128,uint128,uint128[]) (runs: 256, : 4423799, ~:
↪ 4416458)
[PASS] testRegisterIdentitiesWithCorrectInputsFromKnown() (gas: 5404561)
[PASS] testRegisterIdentitiesWithRunsOfZeroes(uint8,uint8) (runs: 256, : 4385219, ~: 4406340)
Test result: ok. 17 passed; 0 failed; finished in 12.83s
```

9.2.2 State Bridge

```
> make test
FOUNDRY_PROFILE=debug forge test
[] Compiling...
[] Compiling 1 files with 0.8.15
[] Solc 0.8.15 finished in 789.39ms
Compiler run successful (with warnings)
warning[6321]: Warning: Unnamed return variable can remain unassigned. Add an explicit return with value to all
↪  non-reverting code paths or name the variable.
  --> lib/contracts/contracts/lib/MerklePatriciaProof.sol:20:30:
   |
20 |     ) internal pure returns (bool) {
   |                              ^^^^



warning[5667]: Warning: Unused function parameter. Remove or comment out the variable name to silence this warning.
  --> src/mock/WorldIDIdentityManagerMock.sol:25:29:
   |
25 |     function checkValidRoot(uint256 root) public view returns (bool) {
   |                             ^^^^^^^^^^^^



warning[2018]: Warning: Function state mutability can be restricted to pure
  --> src/mock/WorldIDIdentityManagerMock.sol:25:5:
   |
25 |     function checkValidRoot(uint256 root) public view returns (bool) {
   |     ^ (Relevant source part starts here and spans across multiple lines).




Running 2 tests for src/test/PolygonWorldID.t.sol:PolygonWorldIDTest
[PASS] testCanGetTreeDepth(uint8) (runs: 256, : 7017617, ~: 7017617)
[PASS] testConstructorWithInvalidTreeDepth(uint8) (runs: 256, : 6333465, ~: 6333486)
Test result: ok. 2 passed; 0 failed; finished in 274.27ms

Running 7 tests for src/test/OpWorldID.t.sol:OpWorldIDTest
[PASS] testCanGetTreeDepth(uint8) (runs: 256, : 7028161, ~: 7028161)
[PASS] testConstructorWithInvalidTreeDepth(uint8) (runs: 256, : 6353495, ~: 6353514)
[PASS] test_expiredRoot_reverts(uint256,uint256) (runs: 256, : 181496, ~: 181496)
[PASS] test_onlyOwner_notMessenger_reverts(uint256) (runs: 256, : 28747, ~: 28747)
[PASS] test_onlyOwner_notOwner_reverts(uint256) (runs: 256, : 32443, ~: 32443)
[PASS] test_receiveVerifyInvalidRoot_reverts(uint256) (runs: 256, : 122050, ~: 123077)
[PASS] test_receiveVerifyRoot_succeeds(uint256) (runs: 256, : 117204, ~: 118402)
Test result: ok. 7 passed; 0 failed; finished in 309.51ms

Running 7 tests for src/test/StateBridge.t.sol:StateBridgeTest
[PASS] test_canSelectFork_succeeds() (gas: 5639)
[PASS] test_notOwner_transferOwnershipOptimism_reverts(address,address,bool) (runs: 256, : 13915, ~: 13915)
[PASS] test_notOwner_transferOwnership_reverts(address,address) (runs: 256, : 13756, ~: 13756)
[PASS] test_owner_transferOwnershipOptimism_succeeds(address,bool) (runs: 256, : 117769, ~: 117769)
[PASS] test_owner_transferOwnership_succeeds(address) (runs: 256, : 20726, ~: 20726)
[PASS] test_sendRootMultichain_reverts(uint256,address) (runs: 256, : 174298, ~: 174298)
[PASS] test_sendRootMultichain_succeeds(uint256) (runs: 256, : 174563, ~: 174563)
Test result: ok. 7 passed; 0 failed; finished in 197.79s
```

9.2.3 Airdrop Example

```
> make test
FOUNDRY_PROFILE=debug forge test
[] Compiling...
[] Compiling 14 files with 0.8.19
[] Solc 0.8.19 finished in 2.08s
Compiler run successful

Running 4 tests for src/test/WorldIDAirdrop.t.sol:WorldIDAirdropTest
[PASS] testCanClaim(uint256,uint256) (runs: 256, : 95382, ~: 95382)
[PASS] testCannotDoubleClaim(uint256,uint256) (runs: 256, : 100246, ~: 100246)
[PASS] testCannotUpdateAirdropAmountIfNotManager(address) (runs: 256, : 14977, ~: 14977)
[PASS] testUpdateAirdropAmount() (gas: 19462)
Test result: ok. 4 passed; 0 failed; finished in 40.18ms

Running 6 tests for src/test/WorldIDMultiAirdrop.t.sol:WorldIDMultiAirdropTest
[PASS] testCanClaim(uint256,uint256) (runs: 256, : 220864, ~: 220864)
[PASS] testCanCreateAirdrop() (gas: 139154)
[PASS] testCanUpdateAirdropDetails() (gas: 149988)
[PASS] testCannotClaimNonExistantAirdrop(uint256,uint256) (runs: 256, : 50340, ~: 50340)
[PASS] testCannotDoubleClaim(uint256,uint256) (runs: 256, : 223165, ~: 223165)
[PASS] testNonOwnerCannotUpdateAirdropDetails(address) (runs: 256, : 144235, ~: 144235)
Test result: ok. 6 passed; 0 failed; finished in 48.01ms
```

9.3 Code Coverage

9.3.1 World ID

```
> forge coverage
```

The relevant output is presented below.

```
----------------------------------------------|------------------|------------------|-----------------|-----------------|
| File                                        | % Lines          | % Statements     | % Branches      | % Funcs         |
|---------------------------------------------|------------------|------------------|-----------------|-----------------|
| src/WorldIDIdentityManagerImplV1.sol        | 91.82% (101/110) | 92.44% (110/119) | 81.25% (39/48)  | 96.67% (29/30)  |
| src/WorldIDRouterImplV1.sol                 | 97.37% (37/38)   | 97.67% (42/43)   | 100.00% (12/12) | 90.00% (9/10)   |
| src/abstract/WorldIDImpl.sol                | 100.00% (0/0)    | 100.00% (0/0)    | 100.00% (0/0)   | 100.00% (1/1)   |
| src/data/VerifierLookupTable.sol            | 100.00% (10/10)  | 100.00% (10/10)  | 100.00% (4/4)   | 100.00% (5/5)   |
| src/utils/CheckInitialized.sol              | 100.00% (1/1)    | 100.00% (1/1)    | 100.00% (0/0)   | 100.00% (1/1)   |
| src/utils/SemaphoreTreeDepthValidator.sol   | 0.00% (0/3)      | 0.00% (0/3)      | 100.00% (0/0)   | 0.00% (0/1)     |
| src/utils/UnimplementedTreeVerifier.sol     | 0.00% (0/5)      | 0.00% (0/5)      | 100.00% (0/0)   | 0.00% (0/1)     |
| Total                                       | 89.22% (149/167) | 90.05% (163/181) | 85.93% (55/64)  | 91.83% (45/49)  |
```

9.3.2 State Bridge

```
> forge coverage
```

The relevant output is presented below.

```
----------------------------------------------|----------------|-----------------|----------------|----------------|
| File                                        | % Lines        | % Statements    | % Branches     | % Funcs        |
|---------------------------------------------|----------------|-----------------|----------------|----------------|
| src/OpWorldID.sol                           | 50.00% (1/2)   | 50.00% (1/2)    | 100.00% (0/0)  | 50.00% (1/2)   |
| src/PolygonWorldID.sol                      | 0.00% (0/3)    | 0.00% (0/4)     | 100.00% (0/0)  | 0.00% (0/2)    |
| src/StateBridge.sol                         | 0.00% (0/16)   | 0.00% (0/18)    | 0.00% (0/2)    | 0.00% (0/5)    |
| src/abstract/WorldIDBridge.sol              | 63.64% (14/22) | 63.64% (14/22)  | 50.00% (6/12)  | 42.86% (3/7)   |
| src/utils/SemaphoreTreeDepthValidator.sol   | 0.00% (0/3)    | 0.00% (0/3)     | 100.00% (0/0)  | 0.00% (0/1)    |
| Total                                       | 32.60% (15/46) | 30.61% (15/49)  | 42.85% (6/14)  | 23.52% (4/17)  |
```

9.3.3 Airdrop Example

```
> forge coverage
```

The relevant output is presented below.

```
-------------------------------|-----------------|-----------------|-----------------|---------------|
| File                         | % Lines         | % Statements    | % Branches      | % Funcs       |
|------------------------------|-----------------|-----------------|-----------------|---------------|
| src/WorldIDAirdrop.sol       | 100.00% (7/7)   | 100.00% (8/8)   | 100.00% (4/4)   | 100.00% (2/2) |
| src/WorldIDMultiAirdrop.sol  | 100.00% (14/14) | 100.00% (17/17) | 100.00% (6/6)   | 100.00% (3/3) |
| Total                        | 100.00% (21/21) | 100.00% (25/25) | 100.00% (10/10) | 100.00% (5/5) |
```

9.4 Slither

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

10 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