Veridise. Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Semaphore

Version 3.0

Veridise Inc.

January 5, 2023

► **Prepared For:**

Semaphore
https://semaphore.appliedzkp.org/

► **Prepared By:**

Jon Stephens
Shankara Pailoor
Xiangan He
Daniel Domínguez Álvarez
Hanzhi Liu
► **Contact Us:** contact@veridise.com

► **Version History:**

Jan 5, 2023 V1

Contents

Contents iii

1 Executive Summary 1

2 Project Dashboard 3

3 Audit Goals and Scope 5
3.1 Audit Goals . . . 5
3.2 Audit Methodology & Scope . . . 5
3.3 Classification of Vulnerabilities . . . 6

4 Vulnerability Report 7
4.1 Detailed Description of Bugs . . . 8
4.1.1 V-SEM-VUL-001: No Zero Value Validation . . . 8
4.1.2 V-SEM-VUL-002: New Leaf may be Larger than Prime Field on Update . . . 9
4.1.3 V-SEM-VUL-003: Editor's Entity may be Overwritten . . . 10
4.1.4 V-SEM-VUL-004: Identity Commitment not Unique to Group . . . 11
4.1.5 V-SEM-VUL-005: Index Calculation can be Manipulated . . . 12
4.1.6 V-SEM-VUL-006: Infinite Loop if Input Array is too Large . . . 13
4.1.7 V-SEM-VUL-007: merkleRootDuration cannot be changed . . . 14
4.1.8 V-SEM-VUL-008: merkleRootDuration Could Allow Removed Member to Prove they are in Group . . . 15
4.1.9 V-SEM-VUL-009: nLevels not Constrained in Circuit . . . 16
4.1.10 V-SEM-VUL-010: Different Checks used to Determine if Group Exists . . . 17
4.1.11 V-SEM-VUL-011: No Check if Member Exists . . . 18
4.1.12 V-SEM-VUL-012: Hash Collision could break merkle tree . . . 19

5 Formal Verification 21
5.1 Detailed Description of Formal Verification Results . . . 22
5.1.1 V-SEM-SPEC-001: CalculateSecret Functional Correctness . . . 22
5.1.2 V-SEM-SPEC-002: CalculateIdentityCommitment Functional Correctness 23
5.1.3 V-SEM-SPEC-003: CalculateNullifierHash Functional Correctness . . . 24
5.1.4 V-SEM-SPEC-004: Semaphore Functional Correctness . . . 25
5.1.5 V-SEM-SPEC-005: MerkleTreeInclusionProof Functional Correctness . . . 27
5.1.6 V-SEM-SPEC-006: MultiMux1 Functional Correctness . . . 30
5.1.7 V-SEM-SPEC-007: Poseidon is Deterministic . . . 32

1 Executive Summary

From Dec. 1 to Dec. 31, Semaphore engaged Veridise to review the security of their Groups v3. The review covered the Zero-Knowledge circuits and on-chain contracts that implement the protocol logic. Veridise conducted the assessment over 16 person-weeks, with 4 engineers reviewing code over 4 weeks on commit `27320f1`. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers as well as extensive manual auditing.

Summary of issues detected. The audit uncovered 12 issues, 2 of which are assessed to be of high or critical severity by the Veridise auditors. Specifically V-SEM-VUL-001 gives a group creator implicit and permanent access to participate in a group and V-SEM-VUL-002 allows an admin to update a leaf with a value larger than the snark scalar field. The veridise auditors also identified several moderate-severity issues, including the ability to overwrite a whistleblower group's admin (V-SEM-VUL-003), the ability to manipulate the calculated index of an update to the merkle tree (V-SEM-VUL-005) and an infinite loop if the admin attempts to add too many members at once (V-SEM-VUL-006).

Code assessment. The Semaphore protocol provides a method of privately proving one's membership in a group. It does so by allowing users to create a unique cryptographic identifier, or identity commitment, that they generate from two secret values. This identity commitment is added to an incremental merkle tree, which is used to represent the group and its members. Once added, members can prove their membership in the group by interacting with Semaphore's ZK Circuit. The circuit takes as input a user's two secret values, the proof of membership in the merkle tree, the group's external nullifier and the public value the user wishes to share. It then generates a proof of the user's membership which can be submitted to Semaphore's contracts for validation and verification. Semaphore also provides two sample applications to demonstrate how Groups v3 can be used by other developers.

Semaphore provided the source code for the protocol, which includes the zk-circuits and smart contracts. A hardhat-based test-suite accompanied the source-code with tests written by the developers. Finally, the developers also provided the documentation for the previous version of the protocol, much of which still applies to v3.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Semaphore Groups v3	`27320f1`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Dec. 1 - Dec. 31, 2022	Manual & Tools	4	16 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Resolved
Critical-Severity Issues	0	0
High-Severity Issues	2	2
Medium-Severity Issues	3	3
Low-Severity Issues	2	2
Warning-Severity Issues	4	4
Informational-Severity Issues	1	1
TOTAL	12	12

Table 2.4: Category Breakdown.

Name	Number
Data Validation	3
Access Control	2
Usability	2
Bad Randomness	1
Logic Error	1
Denial of Service	1
Maintainability	1
Hash Collision	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of Semaphore's ZK circuits and the Groups v3 smart contracts. In our audit, we sought to answer the following questions:

- Are the constraints defined by the ZK circuit properly constrained?
- Can a user manipulate the public inputs or outputs of the circuit?
- Is it possible for a user to prove their membership in the group without being added?
- If a user is added to a group, can they be removed?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a combination of human experts and automated program analysis & testing tools. In particular, we conducted our audit with the aid of the following techniques:

- *Static analysis.* To identify potential common vulnerabilities, we leveraged our custom smart contract analysis tool Vanguard, as well as the open-source tool Slither. These tools are designed to find instances of common smart contract vulnerabilities, such as reentrancy and uninitialized variables.
- *Fuzzing/Property-based Testing.* We also leverage fuzz testing to determine if the protocol may deviate from the expected behavior. To do this, we formalize the desired behavior of the protocol as [V] specifications and then use our fuzzing framework OrCa to determine if a violation of the specification can be found.
- *Formal Verification.* To prove the correctness of the ZK circuits we used Coda, our formal verification project based on the Coq interactive theorem prover. To do this, we formalized the intended behavior of the Circom templates and then proved the correctness of the implementation with respect to the formalized specifications.

Scope. This audit reviewed the ZK circuits and on-chain behaviors of Semaphore Groups v3. As such, Veridise auditors first inspected the provided tests and documentation to better understand the desired behavior of the provided source code at a more granular level. They then began a multi-week manual audit of the code assisted by both static analyzers and automated testing. Finally, they formalized the intended behavior of the Semaphore circuit and formally verified it with the help of Coda.

In terms of the audit, the key components include the following:

- The main Semaphore contract
- The zk-kit Incremental Merkle Tree implementation
- The Semaphore whistleblowing and voting extensions
- The Semaphore ZK circuit

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

In this case, we judge the likelihood of a vulnerability as follows:

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

In addition, we judge the impact of a vulnerability as follows:

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowleged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-SEM-VUL-001	No Zero Value Validation	High	Fixed
V-SEM-VUL-002	Update Leaf may be Larger than Prime Field	High	Fixed
V-SEM-VUL-003	Editor's Entity may be Overwritten	Medium	Fixed
V-SEM-VUL-004	Identity Commitment not Unique to Group	Medium	Intended Behavior
V-SEM-VUL-005	Index Calculation can be Manipulated	Medium	Fixed
V-SEM-VUL-006	Infinite Loop if Input Array is too Large	Low	Fixed
V-SEM-VUL-007	merkleRootDuration cannot be changed	Low	Fixed
V-SEM-VUL-008	Removed Member may Prove Group Membership	Warning	Intended Behavior
V-SEM-VUL-009	nLevels not Constrained in Circuit	Warning	Fixed
V-SEM-VUL-010	Multiple Different Group Existence Checks	Warning	Fixed
V-SEM-VUL-011	No Check if Member Exists	Warning	Intended Behavior
V-SEM-VUL-012	Hash Collision could break merkle tree	Info	Acknowledged

4.1 Detailed Description of Bugs

4.1.1 V-SEM-VUL-001: No Zero Value Validation

Severity	High	**Commit**	27320f1
Type	Access Control	**Status**	Fixed
Files	SemaphoreGroups.sol, semaphore.circom		
Functions	N/A		

In the Semaphore protocol groups are backed by incremental merkle trees. Unlike merkle trees which are static, incremental merkle trees allow the tree to be modified. This is done by manipulating the leaves of the tree with a special `zeroValue` that indicates a leaf is empty (since incremental merkle trees are complete binary trees). In the Semaphore protocol, the `zeroValue` appears to be an implicit member a the group as one can prove they belong to the group if they know the `identityNullifier` and `identityTrapdoor` of the `zeroValue`. This implicit member, however, has a few properties that are not shared by others:

1. It cannot be removed from the group as removing a member replaces the leaf's value with the `zeroValue`. This seems to violate an invariant that an added member should be removable
2. A `MemberAdded` event is not emitted to indicate its membership in the group

Impact First, this value allows the creator of a group guaranteed access to the group. In certain circumstances this may be undesired (for example if the admin is not the group creator such as if the admin is a DAO that votes on who to add/remove or if an admin is changed) as the original creator has a permanent method of influencing the application that uses the groups. There are similar methods an admin (who might not be the group creator) can use without the `zeroValue` but these (1) are more visible as adding members is a matter of public record and (2) can be undone by removing the user.

Second, if common values such as `0` are repeatedly used and the identity commitment of this value is eventually compromised, such a user would be able to gain membership to all groups that use this value as the `zeroValue`.

Recommendation Specifically disallow proofs where the leaf corresponds to the `zeroValue`. As a result, users should not be able to call `addMember` or `updateMember` where the member to add or member to update is the `zeroValue` to prevent possible usability issues (i.e. in case the added/updated member is intended to be a real user).

4.1.2 V-SEM-VUL-002: New Leaf may be Larger than Prime Field on Update

Severity	High	**Commit**	27320f1
Type	Data Validation	**Status**	Fixed
Files	SemaphoreGroups.sol		
Functions	_updateMember		

`Semaphore.sol` allows admins to update users' commitments through the external call `updateMembers`. However, there is no validation done in the call stack to ensure that the new commitment is less than the snark field order.

In more detail, `updateMember` (line 106 in `Semaphore.sol`) is is a thin wrapper around `_updateMember` in `SemaphoreGroups.sol` (line 58) which in turn calls the `update` function in `zk-kit`'s `IncrementalBinaryTree.sol` (line 69). There is no validation in any of these calls: unlike `insert` in `IncrementalBinaryTree.sol`, `update` does not check that `newLeaf < SNARK_SCALAR_FIELD`.

Impact If a user manages to get their commitment updated to a value larger than the SNARK field order, then neither that commitment nor its sibling in the tree can be updated or removed. This is because updating or removing a commitment to a group internally calls the `verify` function in `zk-kit` which checks if the sibling of the commitment getting modified is less than the SNARK field. However, since the sibling will be larger than the SNARK field, the call will revert (line 165 `IncrementalBinaryTree.sol`).

Recommendation We recommend that `Semaphore.sol` require that commitments are smaller than the SNARK field in both `insert` and `update`.

4.1.3 V-SEM-VUL-003: Editor's Entity may be Overwritten

Severity	Medium	**Commit**	27320f1
Type	Data Validation	**Status**	Fixed
Files	SemaphoreWhistleblowing.sol		
Functions	createEntity		

The whistleblowing extension allows entities to be created for whistleblowers that allow them to leak information anonymously. When a new entity is created, the `SemaphoreWhistleblowing` contract creates a group that corresponds to the entity. It then associates the entity's editor with the created group via the group id. Currently, however, no validation is performed on the `entities` mapping to determine if the editor is already associated with an `entityId`. As a result, an editor's associated entity could be overwritten accidentally or maliciously.

Impact If an entity's editor is overwritten, that entity would no longer be able to add or remove whistleblowers in the future. A malicious actor could therefore use `createEntity` to disrupt the expected operation of the contract.

Recommendation Either allow an editor to be associated with multiple entities or check if the editor is already associated with an entity.

4.1.4 V-SEM-VUL-004: Identity Commitment not Unique to Group

Severity	Medium	**Commit**	27320f1
Type	Bad Randomness	**Status**	Intended Behavior
Files	semaphore.circom		
Functions	CalculateIdentityCommitment		

A Semaphore identity commitment is computed using two secret that an owner must know: an identity trapdoor and an identity nullifier. Since all of the information used to determine the identity commitment is controlled by the user, the following is possible:

1. If the same trapdoor and nullifier is used to compute the identity commitment for multiple groups, the same member will be added, allowing information to be learned by about this member based on the groups they join.
2. If the identity commitment is compromised for a single group (either due to data theft or an unlikely hash collision) all groups that the user participates in with these values are compromised because the identity commitment of a member is a matter of public record. Thus an attacker can find all groups that have added the given identity commitment and prove their membership of those two groups.

Impact By using the same information to compute the identity commitment, many semaphore groups can be easily compromised if a single identity commitment is compromised simply by inspecting group membership.

Recommendation Include information unique to a group such as the `groupId` in the computation of an identity commitment to increase the effort of compromising a group in the event of theft or luck.

Developer Response The Semaphore authors allow developers to decide if they want to enforce unique identity commitments or not. Developers who want to ensure that the identity commitments are unique can include a unique group identifier (such as `groupId`) in the identity generation process using the provided JS library.

4.1.5 V-SEM-VUL-005: Index Calculation can be Manipulated

Severity	Medium	**Commit**	27320f1
Type	Logic Error	**Status**	Fixed
Files	SemaphoreGroups.sol		
Functions	_updateMember, proofPathIndicesToMemberIndex		

When updating the Merkle Tree, the index of the updated member is calculated to validate that the updated member is within the subtree of added members. Since `proofPathIndices` is a `uint8`, however, it is possible to update a member past `numberOfLeaves` (or rather the last index at which a user has been added). To do so, a user can replace all 1s with 2s to cause the index calculation to report that index 0 was updated. This occurs both in SemaphoreGroup's `proofPathIndicesToMemberIndex` function and zk-kit's `update` function.

Impact Using this technique the admin can make it appear as though they updated a particular index rather than adding a member after `numberOfLeaves`. This could be used to ensure that an updated member gets overwritten in the future.

Recommendation Enforce that `proofPathIndices` must be binary (either by making this an array of booleans or requiring that an entry be 1 or 0).

4.1.6 V-SEM-VUL-006: Infinite Loop if Input Array is too Large

Severity	Low	**Commit**	27320f1
Type	Denial of Service	**Status**	Fixed
Files	Semaphore.sol		
Functions	addMembers		

The `Semaphore` contract allows the admin to add group members in a batch by calling `addMembers` (presumably for gas efficiency). When this function is called, a for loop iterates over all `identityCommitments` upon which `_addMember` is called. This for loop, however, uses a `uint8` as the iterator and makes the iterator increment unchecked. As a result, if the input array's size is larger than 255 (the maximum value of a uint8) then the iterator's value will overflow causing the loop to restart at 0 resulting in an infinite loop.

Impact If an admin adds more than 255 members, the infinite loop will consume all of the transaction's gas and then revert. This therefore can waste a user's funds

Recommendation Either ensure that the admin adds less than 256 members or increase the size of the iterator variable.

4.1.7 V-SEM-VUL-007: merkleRootDuration cannot be changed

Severity	Low	**Commit**	27320f1
Type	Usability	**Status**	Fixed
Files	Semaphore.sol		
Functions	N/A		

When the merkle root of a group's incremental merkle tree is updated, the `Semaphore` contract allows the old root to still be used to verify proofs as long as it is within `merkleRootDuration` of the root's creation. The `merkleRootDuration` is set when a new group is created. However, no validation is performed on `merkleRootDuration` which could lead to issues such as `verifyProof` reverting due to an overflow if the value is too large.

Impact The admin might not know an appropriate value for the `merkleRootDuration` and may like to change it in the the initial value is inconvenient. In addition, under certain circumstances a poorly chosen value could cause `verifyProof` to fail.

Recommendation Allow the group admin to change the `merkleRootDuration` if desired.

4.1.8 V-SEM-VUL-008: merkleRootDuration Could Allow Removed Member to Prove they are in Group

Severity	Warning	**Commit**	27320f1
Type	Access Control	**Status**	Intended Behavior
Files	Semaphore.sol		
Functions	verifyProof		

The semaphore protocol allows multiple Merkle roots to be used when generating membership proofs. In particular, each group has an associated grace period (`merkleRootDuration`) such that all roots created within `merkleRootDuration` of the transaction can be used to perform the check. This is seen in the following snippet from `verifyProof` in `Semaphore.sol`:

```
if (block.timestamp > merkleRootCreationDate + merkleRootDuration) {
    revert Semaphore__MerkleTreeRootIsExpired();
}
```

A consequence of this is that a removed member can submit valid membership proofs.

Impact The impact of this will depend on the application in question. For example, an anonymous voting protocol built on `Semaphore.sol` will allow removed members to vote as long as they vote within an hour of being removed.

Recommendation We recommend that the implementation does not allow roots corresponding to removals or updates be used or specific grace periods for such cases.

Developer Response This feature was added to address some usability concerns. With the ability to change the grace period, it should be possible for an admin to decide whether they want to temporarily adjust the duration if they want to ensure a removed member doesn't prove their membership.

4.1.9 V-SEM-VUL-009: nLevels not Constrained in Circuit

Severity	Warning	**Commit**	27320f1
Type	Usability	**Status**	Fixed
Files	semaphore.circom		
Functions	Semaphore		

Currently a comment exists within `semaphore.circom` stating that `nLevels < 32` and the Semaphore smart contracts require `depth <= 32` and `depth >= 16`. To ensure users are instantiating the circuit correctly, if there are constraints on `nLevels` they should be asserted in the circuit.

Impact A user can instantiate the circuit with a size that will not be accepted by Semaphore.

Recommendation If constraints exist on `nLevels` add them as an assertion.

4.1.10 V-SEM-VUL-010: Different Checks used to Determine if Group Exists

Severity	Warning	**Commit**	27320f1
Type	Maintainability	**Status**	Fixed
Files	SemaphoreGroups.sol, Semaphore.sol		
Functions	Multiple		

The Semaphore protocol allows users to create, update, and verify membership to groups via several APIs such as `updateMember`, `createGroup`, `removeMember`, etc. Each function checks whether the group getting created/modified exists. We observed two different checks used:

```
if (getMerkleTreeRoot(groupId) == 0) {
    revert Semaphore__GroupDoesNotExist();
}
```

as well as:

```
if (getMerkleTreeDepth(groupId) == 0) {
    revert Semaphore__GroupDoesNotExist();
}
```

The former is used in `verifyProof` (`Semaphore.sol`) as well as `_updateMember` and `_removeMember` in `SemaphoreGroups.sol`. The latter is used in `_createGroup` and `_addMember` in `SemaphoreGroups.sol`.

In the overwhelming majority of cases, they should both return 0 iff the group doesn't exist. However, `getMerkleTreeRoot` just returns the root of the `MerkleTree` which *could* be 0 after taking the hashes of all the leaves. As such, it is possible that this check fails when the group actually exists.

Impact In the unlikely scenario that the group exists and the root hash is 0, legitimate `verify`, `update` , and `remove` transactions would get rejected until the root hash changes.

Recommendation We recommend that `getMerkleTreeDepth` be used everywhere as the check since it is just as efficient and not susceptible to this corner case (however unlikely it may be).

4.1.11 V-SEM-VUL-011: No Check if Member Exists

Severity	Warning	**Commit**	27320f1
Type	Data Validation	**Status**	Intended Behavior
Files	SemaphoreGroups.sol		
Functions	_addMember, _updateMember		

When a a member is added to a group or a member is updated, no validation is performed to determine if identity commitment already has membership.

Impact If an individual is added to a group multiple times, it may be difficult to revoke their membership as each entry in the tree must be revoked individually.

Recommendation Check if the added member is already a member of the group.

Developer Response The developers indicate that they expect the off-chain application to perform such checks rather than performing them on-chain since the leaves are not stored on-chain.

4.1.12 V-SEM-VUL-012: Hash Collision could break merkle tree

Severity	Info	**Commit**	27320f1
Type	Hash Collision	**Status**	Acknowledged
Files	zk-kit::IncrementalBinaryTree.sol		
Functions	update		

When performing an update of the incremental binary merkle tree, it is possible for nodes along the path of the "last subtree" to be modified. This "last subtree" corresponds to the "rightmost" node of a particular level of the tree that must be modified when additions are made. To determine if one of these nodes must be updated, the developers check if the computed hash for the current level corresponds to the hash of the last subtree for that level. If the two hashes match, then the last subtree hash is overwritten. While the chances of a collision are extremely low, if one were to occur and, the resulting modification of the last subtree node would break the incremental merkle tree such that it cannot be updated in the future.

```
    function update(
        IncrementalTreeData storage self,
        uint256 leaf,
        uint256 newLeaf,
        uint256[] calldata proofSiblings,
        uint8[] calldata proofPathIndices
    ) public {
        ...

        for (uint8 i = 0; i < depth; ) {
            updateIndex |= uint256(proofPathIndices[i] & 1) << uint256(i);
            if (proofPathIndices[i] == 0) {
                if (proofSiblings[i] == self.lastSubtrees[i][1]) {
                    self.lastSubtrees[i][0] = hash;
                }

                hash = PoseidonT3.poseidon([hash, proofSiblings[i]]);
            } else {
                if (proofSiblings[i] == self.lastSubtrees[i][0]) {
                    self.lastSubtrees[i][1] = hash;
                }

                hash = PoseidonT3.poseidon([proofSiblings[i], hash]);
            }

            unchecked {
                ++i;
            }
        }

        ...
    }
```

Snippet 4.1: The snippet of zk-kit's update function where a hash collision could cause the tree to become malformed

Recommendation The same check could be performed using the `updateIndex` calculation, but the chances of a collision occurring are extremely low.

5 Formal Verification

In this section, we describe the specifications that were used to formally verify the correctness of the ZK circuits. For each specification, we log its current status (i.e. verified, not verified). Table 5.1 summarizes the specifications and their verification status:

Table 5.1: Summary of Discovered Vulnerabilities.

ID	Description	Status
V-SEM-SPEC-001	CalculateSecret Functional Correctness	Verified
V-SEM-SPEC-002	CalculateIdentityCommitment Functional Correctness	Verified
V-SEM-SPEC-003	CalculateNullifierHash Functional Correctness	Verified
V-SEM-SPEC-004	Semaphore Functional Correctness	Verified
V-SEM-SPEC-005	MerkleTreeInclusionProof Functional Correctness	Verified
V-SEM-SPEC-006	MultiMux1 Functional Correctness	Verified
V-SEM-SPEC-007	Poseidon is Deterministic	Verified

5.1 Detailed Description of Formal Verification Results

5.1.1 V-SEM-SPEC-001: CalculateSecret Functional Correctness

Commit	27320f1	**Status**	Verified
Files	semaphore.circom		
Functions	CalculateSecret		

Description The output of the circuit is the Poseidon hash of the two input arguments.

Informal Specification

$$\text{out} = \text{poseidon}_2(\text{identityNullifier}, \text{identityTrapdoor})$$

Formal Definition The following shows the formal definition for the CalculateSecret template:

```
Definition cons (identityNullifier identityTrapdoor: F) (out: F) :=
  exists poseidon: @Poseidon.t 2,
    poseidon.(Poseidon.inputs)[0] = identityNullifier /\
    poseidon.(Poseidon.inputs)[1] = identityTrapdoor /\
    out = poseidon.(Poseidon.out).
```

Formal Specification The following shows the formal specification for the CalculateSecret template:

```
Definition spec (c: t) : Prop :=
  forall x y,
  c.(identityNullifier) = x ->
  c.(identityTrapdoor) = y ->
  c.(out) = poseidon_2 x y.
```

Proof The following shows the soundness proof for the CalculateSecret template:

```
Theorem soundness:
  forall (c: t), spec c.
Proof.
  unwrap_C. unfold spec. intros. destruct c. subst. simpl in *.
  destruct _cons0 as [x _cons]. destruct _cons. destruct H0.
  subst. pose proof Poseidon.PoseidonHypo.poseidon_2_spec. erewrite H;eauto.
Qed.
```

5.1.2 V-SEM-SPEC-002: CalculateIdentityCommitment Functional Correctness

Commit	27320f1	**Status**	Verified
Files	semaphore.circom		
Functions	CalculateIdentityCommitment		

Description The output of the circuit is the Poseidon hash of the input argument.

Informal Specification

$$\text{out} = \text{poseidon}_1(\text{secret})$$

Formal Definition The following shows the formal definition for the CalculateIdentityCommitment template:

```
Definition cons (secret: F) (out: F) :=
  exists poseidon: @Poseidon.t 1,
    poseidon.(Poseidon.inputs)[0] = secret /\
    out = poseidon.(Poseidon.out).
```

Formal Specification The following shows the formal specification for the CalculateIdentityCommitment template:

```
Definition spec (c: t) : Prop :=
  forall x,
  c.(secret) = x ->
  c.(out) = poseidon_1 x.
```

Proof The following shows the soundness proof for the CalculateIdentityCommitment template:

```
Theorem soundness:
  forall (c: t), spec c.
Proof.
  unwrap_C. unfold spec. intros. destruct c. subst. simpl in *.
  destruct _cons0 as [x _cons]. destruct _cons.
  subst. pose proof Poseidon.PoseidonHypo.poseidon_1_spec. erewrite H;eauto.
Qed.
```


5.1.3 V-SEM-SPEC-003: CalculateNullifierHash Functional Correctness

Commit	27320f1	**Status**	Verified
Files	semaphore.circom		
Functions	CalculateNullifierHash		

Description The output of the circuit is the Poseidon hash of the two input arguments.

Informal Specification

$$\text{out} = \text{poseidon}_2(\text{externalNullifier}, \text{identityNullifier})$$

Formal Definition The following shows the formal definition for the CalculateNullifierHash template:

```
Definition cons (externalNullifier identityNullifier: F) (out: F) :=
  exists poseidon: @Poseidon.t 2,
    poseidon.(Poseidon.inputs)[0] = externalNullifier /\
    poseidon.(Poseidon.inputs)[1] = identityNullifier /\
    out = poseidon.(Poseidon.out).
```

Formal Specification The following shows the formal specification for the CalculateNullifierHash template:

```
Definition spec (c: t) : Prop :=
  forall x y,
  c.(externalNullifier) = x ->
  c.(identityNullifier) = y ->
  c.(out) = poseidon_2 x y.
```

Proof The following shows the soundness proof for the CalculateNullifierHash template:

```
Theorem soundness:
  forall (c: t), spec c.
Proof.
  unwrap_C. unfold spec. intros. destruct c. subst. simpl in *.
  destruct _cons0 as [x _cons]. destruct _cons. destruct H0.
  subst. pose proof Poseidon.PoseidonHypo.poseidon_2_spec. erewrite H;eauto.
Qed.
```

5.1.4 V-SEM-SPEC-004: Semaphore Functional Correctness

Commit	27320f1	**Status**	Verified
Files	semaphore.circom		
Functions	Semaphore		

Description The circuit takes as input a user's secret identity trapdoor and identity nullifier in addition to the group's external nullifier and the proof of inclusion in the merkle tree. Given this information, the circuit will output the root of the merkle tree that contains the given identity commitment and the nullifier hash, which is the Poseidon hash of the user's identity nullifier together with the group's identity nullifier.

Informal Specification

$$\text{nullifierHash} = \text{poseidon}_2(\text{externalNullifier}, \text{identityNullifier})$$

$$\text{identityCommitment} = \text{poseidon}_1(\text{poseidon}_2(\text{identityNullifier}, \text{identityTrapdoor}))$$

$$\text{hashes}(i) = \begin{cases} \text{poseidon}_2(\text{hashes}(i-1), \text{siblings}[i]) & \text{if pathIndices}[i] = 0 \\ \text{poseidon}_2(\text{siblings}[i], \text{hashes}(i-1)) & \text{if pathIndices}[i] = 1 \\ \text{identityCommitment} & \text{if } i = 0 \end{cases}$$

$$\text{root} = \text{hashes}(\text{nLevels})$$

Formal Definition The following shows the formal definition for the Semaphore template:

```
Definition cons (identityNullifier: F) (identityTrapdoor: F) (treePathIndices: F^
    nLevels)
    (treeSiblings: F^nLevels) (signalHash: F) (externalNullifier: F)
    (root: F) (nullifierHash: F) :=
  exists (calculateSecret: CalculateSecret.t) (calculateIdentityCommitment:
    CalculateIdentityCommitment.t) (calculateNullifierHash: CalculateNullifierHash.t)
     (inclusionProof: @MerkleTreeInclusionProof.t nLevels),
  calculateSecret.(CalculateSecret.identityNullifier) = identityNullifier /\
  calculateSecret.(CalculateSecret.identityTrapdoor) = identityTrapdoor /\
  calculateIdentityCommitment.(CalculateIdentityCommitment.secret) = calculateSecret
    .(CalculateSecret.out) /\
  calculateNullifierHash.(CalculateNullifierHash.externalNullifier) =
    externalNullifier /\
  calculateNullifierHash.(CalculateNullifierHash.identityNullifier) =
    identityNullifier /\
  inclusionProof.(MerkleTreeInclusionProof.leaf) = calculateIdentityCommitment.(
    CalculateIdentityCommitment.out) /\
  inclusionProof.(MerkleTreeInclusionProof.pathIndices) = treePathIndices /\
  inclusionProof.(MerkleTreeInclusionProof.siblings) = treeSiblings /\
  root = inclusionProof.(MerkleTreeInclusionProof.root) /\
  nullifierHash = calculateNullifierHash.(CalculateNullifierHash.out) /\
  signalHash * signalHash = signalHash.
```


Formal Specification The following shows the formal specification for the Semaphore template:

```
Definition spec (c: t) : Prop :=
  c.(nullifierHash) = poseidon_2 c.(externalNullifier) c.(identityNullifier) /\
  let identityCommitment := poseidon_1 (poseidon_2 c.(identityNullifier) c.(
    identityTrapdoor)) in
  c.(root) = fold_left (fun (y:F) (x:(F*F)) => if dec (fst x = 0) then (poseidon_2 y
    (snd x)) else (poseidon_2 (snd x) y))
      (combine ('(c.(treePathIndices))) ('(c.(treeSiblings)))) identityCommitment.
```

Proof The following shows the soundness proof for the Semaphore template:

```
Theorem soundness:
  forall (c: t), spec c.
Proof.
  unwrap_C.
  intros c.
  destruct c as [identityNullifier identityTrapdoor treePathIndices treeSiblings
    signalHash externalNullifier root nullifierHash _cons].
  unfold spec, cons in *. simpl.
  destruct _cons as [calculateSecret _cons]. destruct _cons as [
    calculateIdentityCommitment _cons]. destruct _cons as [calculateNullifierHash
    _cons].
  destruct _cons as [inclusionProof _cons]. destruct _cons as [_cons1 _cons2].
    destruct _cons2 as [_cons2 _cons3].
  destruct _cons3 as [_cons3 _cons4]. destruct _cons4 as [_cons4 _cons5]. destruct
    _cons5 as [_cons5 _cons6].
  destruct _cons6 as [_cons6 _cons7]. destruct _cons7 as [_cons7 _cons8]. destruct
    _cons8 as [_cons8 _cons9].
  destruct _cons9 as [_cons9 _cons10]. destruct _cons10 as [_cons10 _cons11]. subst.
  pose proof (CalculateSecret.soundness calculateSecret). pose proof (
    CalculateIdentityCommitment.soundness calculateIdentityCommitment).
  pose proof (CalculateNullifierHash.soundness calculateNullifierHash). pose proof (
    MerkleTreeInclusionProof.soundness inclusionProof).
  unfold CalculateSecret.spec, CalculateIdentityCommitment.spec,
    CalculateNullifierHash.spec, MerkleTreeInclusionProof.spec in *.
  split;auto.
  destruct H2. rewrite H3.
  rewrite _cons6. erewrite H0;eauto. erewrite <- H;auto.
Qed.
```

5.1.5 V-SEM-SPEC-005: MerkleTreeInclusionProof Functional Correctness

Commit	27320f1	**Status**	Verified
Files	tree.circom		
Functions	MerkleTreeInclusionProof		

Description The output of the circuit is the root of the Merkle Tree given the leaf of the tree and its proof of inclusion.

Informal Specification

$$\forall_{0 \le i < \text{nLevels}} i.\ \text{pathIndices}[i] = 0 \lor \text{pathIndices}[i] = 1$$

$$\text{hashes}(i) = \begin{cases} \text{poseidon}_2(\text{hashes}(i-1), \text{siblings}[i]) & \text{if pathIndices}[i] = 0 \\ \text{poseidon}_2(\text{siblings}[i], \text{hashes}(i-1)) & \text{if pathIndices}[i] = 1 \\ \text{leaf} & \text{if } i = 0 \end{cases}$$

$$\text{root} = \text{hashes}(\text{nLevels})$$

Formal Definition The following shows the formal definition for the MerkleTreeInclusionProof template:

```
Definition cons (leaf: F) (pathIndices: F^nLevels) (siblings: F^nLevels) (root: F) :=
  exists (poseidons: (@Poseidon.t 2)^nLevels) (mux: (@MultiMux.t 2)^nLevels)
         (hashes: F^(nLevels + 1)),
  hashes[0] = leaf /\
  let _C :=
    (D.iter (fun i _C =>
    _C /\
    pathIndices[i] * (1 - pathIndices[i]) = 0 /\
    (mux[i].(MultiMux.c))[0][0] = hashes[i] /\
    mux[i].(MultiMux.c)[0][1] = siblings[i] /\
    mux[i].(MultiMux.c)[1][0] = siblings[i] /\
    mux[i].(MultiMux.c)[1][1] = hashes[i] /\
    mux[i].(MultiMux.s) = pathIndices[i] /\
    poseidons[i].(Poseidon.inputs)[0] = mux[i].(MultiMux.out)[0] /\
    poseidons[i].(Poseidon.inputs)[1] = mux[i].(MultiMux.out)[1] /\
    hashes[i + 1] = poseidons[i].(Poseidon.out)
    ) nLevels True) in _C /\
  root = hashes[nLevels].
```

Formal Specification The following shows the formal specification for the MerkleTreeInclusionProof template:

```
Definition spec (c: t) : Prop :=
  (forall i, 0 <= i < nLevels -> binary (c.(pathIndices)[i])) /\
  c.(root) = fold_left (fun (y:F) (x:(F*F)) => if dec (fst x = 0) then (poseidon_2 y
    (snd x)) else (poseidon_2 (snd x) y))
```



```
      (combine ('(c.(pathIndices))) ('(c.(siblings)))) c.(leaf).
```

Proof The following shows the soundness proof for the MerkleTreeInclusionProof template:

```
Lemma fold_left_firstn_S:
  forall (l: list (F*F))(i: nat)(b: F)f,
  i < length l ->
  fold_left f  (l [:S i]) b =
  f (fold_left f (l [:i]) b) (l ! i).
Proof.
  intros.
  assert(l [:S i] = l [:i] ++ ((l ! i)::nil)).
  { erewrite firstn_S;try lia. unfold_default. auto. }
  rewrite H0.
  apply fold_left_app.
Qed.

Lemma combine_fst_n: forall n j (l1 l2: F^n),
  j < n ->
  j < n ->
  fst (combine (' l1) (' l2) ! j) = l1 [j].
Proof.
  intros. pose proof (length_to_list l1). pose proof (length_to_list l2).
  unfold_default. simpl. rewrite combine_nth;simpl;auto.
  rewrite nth_Default_nth_default. rewrite <- nth_default_to_list. unfold_default.
    auto.
  rewrite H1, H2;auto.
Qed.

Lemma combine_snd_n: forall n j (l1 l2: F^n),
  j < n ->
  j < n ->
  snd (combine (' l1) (' l2) ! j) = l2 [j].
Proof.
  intros. pose proof (length_to_list l1). pose proof (length_to_list l2).
  unfold_default. simpl. rewrite combine_nth;simpl;auto.
  rewrite nth_Default_nth_default. rewrite <- nth_default_to_list. unfold_default.
    auto.
  rewrite H1, H2;auto.
Qed.

(* MerkleTreeInclusionProof is sound *)
Theorem soundness:
  forall (c: t), spec c.
Proof.
  unwrap_C.
  intros c.
  destruct c as [leaf pathIndices siblings root _cons].
  unfold spec, cons in *. simpl.
  destruct _cons as [poseidons _cons]. destruct _cons as [mux _cons]. destruct _cons
    as [hashes _cons].
  destruct _cons as [_cons1 _cons2]. destruct _cons2 as [_cons2 _cons3].
  rem_iter. subst. rem_iter.
```

```
47  pose (Inv1 := fun (i: nat) (_cons: Prop) => _cons ->
48      (forall j, j < i -> binary ((pathIndices)[j]))).
49  assert (HInv1: Inv1 nLevels (D.iter f nLevels True)).
50  { apply D.iter_inv; unfold Inv1;intros;try lia.
51    subst. destruct H1. destruct (dec (j0 = j));intuit.
52    + subst. unfold binary.
53      destruct (dec (pathIndices [j] = 0));auto.
54      destruct (dec (pathIndices [j] = 1));auto. fqsatz.
55    + apply H11;auto. lia. }
56  apply HInv1 in _cons2 as inv1.
57  split;intros. apply inv1;lia.
58  pose (Inv2 := fun (i: nat) (_cons: Prop) => _cons ->
59      (hashes [i] = (fold_left
60      (fun (y : F) (x : F * F) => if dec (fst x = 0) then poseidon_2 y (snd x) else
    poseidon_2 (snd x) y)
61      (firstn i (combine (' pathIndices) (' siblings)))
62                (hashes [0])))).
63  assert (HInv2: Inv2 nLevels (D.iter f nLevels True)).
64  { apply D.iter_inv; unfold Inv2;intros;try lia.
65    + simpl. auto.
66    + subst. destruct H1.
67      do 8 destruct H2 as [? H2]. pose proof (MultiMux.soundness (mux [j])). unfold
    MultiMux.spec in H11.
68      erewrite (fold_left_firstn_S (combine (' pathIndices) (' siblings)));simpl.
69      2:{ pose_lengths. rewrite combine_length. rewrite _Hlen4, _Hlen2. lia. }
70      assert(FST: (fst (combine (' pathIndices) (' siblings) ! j) = pathIndices [j]))
    .
71      { rewrite combine_fst_n;auto. }
72      assert(SND: (snd (combine (' pathIndices) (' siblings) ! j) = siblings [j])).
73      { rewrite combine_snd_n;auto. }
74      rewrite FST, SND in *. destruct H11. pose proof (H H1) as HASHJ.
75      destruct (dec (pathIndices [j] = 0)).
76      ++ rewrite e in *. pose proof (H11 H8).
77         rewrite H13 in H9;try lia. rewrite H13 in H10;try lia.
78         rewrite HASHJ in H4. rewrite H4, H6 in *. replace (S j) with (j+1)%nat by
    lia.
79         rewrite H2. apply Poseidon.PoseidonHypo.poseidon_2_spec;auto.
80      ++ pose proof (inv1 j). destruct H13;try lia;try easy. rewrite H13 in *. pose
    proof (H12 H8).
81         rewrite H14 in H9;try lia. rewrite H14 in H10;try lia.
82         rewrite HASHJ in H7. rewrite H5, H7 in *. replace (S j) with (j+1)%nat by
    lia.
83         rewrite H2. apply Poseidon.PoseidonHypo.poseidon_2_spec;auto.  }
84  apply HInv2 in _cons2 as inv2.
85  rewrite inv2. rewrite combine_firstn. pose_lengths.
86  assert((' siblings [:nLevels]) = (' siblings)).
87  { rewrite <- _Hlen1 at 1. apply ListUtil.List.firstn_all. }
88  rewrite <- _Hlen0 at 1. rewrite ListUtil.List.firstn_all. rewrite H. auto.
89 Qed.
```


5.1.6 V-SEM-SPEC-006: MultiMux1 Functional Correctness

Commit	27320f1	**Status**	Verified
Files	mux1.circom		
Functions	MultiMux1		

Description The output of the circuit is equivalent to the first set of inputs if the selector is 0 or the second set of inputs if the selector is 1.

Informal Specification

$$s = 0 \rightarrow \underset{0 \leq i < n}{\forall} i.\ out[i] = c[i][0]$$

$$s = 1 \rightarrow \underset{0 \leq i < n}{\forall} i.\ out[i] = c[i][1]$$

Formal Definition The following shows the formal definition for the MultiMux1 template:

```
Definition cons (c: (F^2)^n) (s: F) (out: F^n) :=
let _C :=
    (D.iter (fun i _C =>
    _C /\
    out[i] = (c[i][1] - c[i][0])*s + c[i][0]
    ) n True) in _C.
```

Formal Specification The following shows the formal specification for the MultiMux1 template:

```
Definition spec (m: t) : Prop :=
  (m.(s) = 0 -> forall i, 0 <= i < n -> m.(out)[i] = m.(c)[i][0]) /\
  (m.(s) = 1 -> forall i, 0 <= i < n -> m.(out)[i] = m.(c)[i][1]).
```

Proof The following shows the soundness proof for the MultiMux1 template:

```
Theorem soundness:
  forall (c: t), spec c.
Proof.
  unwrap_C.
  intros c.
  destruct c as [c s out _cons1].
  unfold spec, cons in *. simpl.
  rem_iter.
  pose (Inv1 := fun (i: nat) (_cons: Prop) => _cons ->
      (forall j, j < i -> out [j] = (c [j ][ 1] - c [j ][ 0]) * s + c [j ][ 0])).
  assert (HInv1: Inv1 n (D.iter f n True)).
  { apply D.iter_inv; unfold Inv1;intros;try lia.
    subst. destruct H1. destruct (dec (j0 = j));intuit.
    + subst. auto.
    + apply H4;auto. lia. }
  apply HInv1 in _cons1 as inv.
```

```
17   split;intros;intuit.
18   - apply inv in H2. subst. fqsatz.
19   - apply inv in H2. subst. fqsatz.
20 Qed.
```

5.1.7 V-SEM-SPEC-007: Poseidon is Deterministic

Commit	27320f1	**Status**	Verified
Files	poseidon.circom		
Functions	Poseidon		

Description The output of the poseidon(1) hash and poseidon(2) hash is deterministic. In other words, given the same inputs, the output of the Poseidon hash must be constant.

Informal Specification

$$\forall\ \text{inputs}, \text{out}_1, \text{out}_2.(\text{poseidon}(\text{inputs}) = \text{out}_1 \land \text{poseidon}(\text{inputs}) = \text{out}_2) \rightarrow \text{out}_1 = \text{out}_2$$

Proof This property was proved using Picus, an in house tool used to verify that ZK circuits are properly constrained. Picus proved Poseidon(1) and Poseidon(2) were properly constrained and since all properly constrained circuits are deterministic, we can safely conclude that Poseidon(1) and Poseidon(2) are deterministic.