Veridise. Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

UniRep Protocol

Veridise Inc.
June 17, 2023

► **Prepared For:**

Unirep
https://developer.unirep.io/

► **Prepared By:**

Jon Stephens
Hanzhi Liu
Xiangan He

► **Contact Us:** contact@veridise.com

► **Version History:**

May 15, 2023	Initial Draft

Contents

Executive Summary 1

From April 17, 2023 to May 19, 2023, Unirep engaged Veridise to review the security of their UniRep Protocol. The review covered the protocol's Zero-Knowledge circuits, on-chain contracts and client-side typescript library. Veridise conducted the assessment over 15 person-weeks, with 3 engineers reviewing code over 5 weeks on commit `0x0985a28`. Due to vulnerabilities found during the course of the audit, the formal verification was performed on commit `0x510c971` as the buggy implementation could not be formally verified. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers as well as extensive manual auditing. In parallel, the Veridise engineers also formally verified that the UniRep Protocol circuits adhere to the formal specifications shown in Section 5.

Code assessment. The Unirep developers provided the source code of the UniRep Protocol for review. This included the on-chain contracts, zero-knowledge circuits and client-side framework code. All of these components form a non-repudiable attestation system that allows applications (or attesters) to assign reputation, among other private data, to private identities. To do so, first an attester must register their application with the UniRep Protocol to initialize the necessary state. Attesters may then register users by providing their public identity (a semaphore commitment) and an attestation of their initial state. With this, users may generate some number of *private identities* per epoch that are cryptographically generated using private and public information about the user, attester and epoch. Such private identities may then receive *attestations* from the application to change the private data of the associated user, including their reputation. After the completion of an epoch, users must then incorporate the attestations received on their private identities into their private data so that they may continue to interact with the application. Additionally, users can use the UniRep Protocol's ZK circuits to prove information about their private data, including their reputation thresholds, private identities.

To facilitate the Veridise auditors' understanding of the code, the Unirep developers also provided blog posts that document the high level design of the protocol and developer documentation describing the low-level components of the protocol. The source code also contained some documentation in the form of READMEs and documentation comments in-line with the source-code. The source code contained a test suite, which covered the individual components of the UniRep Protocol.

Summary of issues detected. The audit uncovered 19 issues, 2 of which are assessed to be of high or critical severity by the Veridise auditors. Specifically, V-UNI-VUL-001 identifies an under-constrained range check that could allow attackers to prove incorrect relationships between values and V-UNI-VUL-002 identifies incorrect uses of the CircomLib's comparison components which could also allow the comparisons to be manipulated. The Veridise auditors also identified several medium-severity issues, including potential overflows (V-UNI-VUL-003, V-UNI-VUL-004) and logical errors (V-UNI-VUL-006, V-UNI-VUL-007) as well as a number of minor issues.

Recommendations. After auditing the protocol, the auditors had a few suggestions to improve the UniRep Protocol. During the audit, Veridise identified several issues that correspond to intended behavior as it is expected that the *applications*, *attesters* or *users* will prevent them. Such issues include:

- The potential for an attester's sum data fields to overflow (V-UNI-VUL-003). Note, that a user's positive and negative reputation are stored in such fields.
- The potential for attestation loss if the state tree is not updated during an epoch (V-UNI-VUL-006).
- The potential for private information leakage through repeated proofs (V-UNI-VUL-015).
- The potential for non-users to submit proofs that are accepted by the protocol via the EpochKeyLite circuit.

We therefore encourage developers building on top of the UniRep Protocol to pay special attention to these issues and to ensure that their protocol is audited before deployment. Additionally, we encourage Unirep to create dedicated pages in their documentation that lists assumptions that they placed on each of these parties as some of the warnings are distributed across the developer documentation.

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
UniRep Protocol	`0x0985a28`	Circom, Solidity, TS	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
April 17 - May 19, 2023	Manual & Tools	3	15 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Resolved
Critical-Severity Issues	2	2
High-Severity Issues	0	0
Medium-Severity Issues	5	5
Low-Severity Issues	5	5
Warning-Severity Issues	7	7
Informational-Severity Issues	0	0
TOTAL	19	19

Table 2.4: Verification Summary.

Type	Number
Functional Correctness	14

Table 2.5: Category Breakdown.

Name	Number
Logic Error	5
Integer Overflow	2
Data Validation	2
Information Leakage	2
Usability Issue	2
Underconstrained Circuit	1
Missing Range Check	1
Unconstrained Public Input	1
Unnecessary Constraints	1
Storage Optimization	1
Access Control	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of the UniRep Protocol's ZK Circuits, on-chain smart contracts and client-side typescript library. In our audit, we sought to answer the following questions:

- Can private information be stolen or leaked?
- Are the circuit signals properly constrained?
- Can a malicious user prove an invalid data relationship?
- Can a user avoid reject or avoid an attestation?
- Are updates to a user's private data consistent with Unirep's documentation?
- Do all public inputs participate in at least one constraint?
- Where appropriate, do the developers validate that signals are within an appropriate range?
- Can a user avoid performing a state transition?

3.2 Audit Methodology & Scope

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

- *Static analysis.* To identify potential common vulnerabilities, we leveraged our custom Zero-Knowledge circuit static analysis tool. This tool is designed to find instances of common vulnerabilities in Zero-Knowledge circuits, such as unused public inputs and dataflow-constraint discrepancies.
- *Formal Verification.* To prove the correctness of the ZK circuits we used a combination of Coda, our formal verification project based on the Coq interactive theorem prover, and Picus, our automated verification tool. 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. The scope of this audit is limited to the following folders in the repository located at `https://github.com/Unirep/Unirep`:

- `/packages/circuits/circuits/`
- `/packages/contracts/contracts/`
- `/packages/utils/src/`
- `/packages/core/src/`

Methodology. The 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 formalized the intended behavior of the UniRep Protocolcircuits and formally verified them with the help of Coda. In parallel, they performed a multi-week manual audit of the code assisted by our static analyzer.

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 in Table 3.2:

Table 3.2: Likelihood Breakdown

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 in Table 3.3:

Table 3.3: Impact Breakdown

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-UNI-VUL-001	Underconstrained Circuit	Critical	Fixed
V-UNI-VUL-002	Missing Range Checks on Comparison Circuits	Critical	Fixed
V-UNI-VUL-003	Potential Overflow in User State Transition	Medium	Intended Behavior
V-UNI-VUL-004	Potential Overflow when Proving Reputation	Medium	Fixed
V-UNI-VUL-005	Malformed ReusableMerkleTree Root	Medium	Fixed
V-UNI-VUL-006	Potential Attestation Loss	Medium	Intended Behavior
V-UNI-VUL-007	Different Definitions of Data Field	Medium	Fixed
V-UNI-VUL-008	Unconstrained Public Input	Low	Fixed
V-UNI-VUL-009	Replacement Data ID Validation not Monotonic	Low	Fixed
V-UNI-VUL-010	Manual Signup may be Incorrect	Low	Fixed
V-UNI-VUL-011	Replacement Data IDs not Monotonic	Low	Fixed
V-UNI-VUL-012	Private Information Stored in Plaintext	Low	Acknowledged
V-UNI-VUL-013	Unnecessary Num2Bits(254) Constraints	Warning	Fixed
V-UNI-VUL-014	Add Template Variable Assertions	Warning	Fixed
V-UNI-VUL-015	Potential for Private Information Leakage	Warning	Acknowledged
V-UNI-VUL-016	Wasted Storage	Warning	Fixed
V-UNI-VUL-017	Assumed Replacement Data Ordering	Warning	Fixed
V-UNI-VUL-018	Confusing Corner Case Logic	Warning	Fixed
V-UNI-VUL-019	Containerize TypeScript Classes	Warning	Fixed

4.1 Detailed Description of Issues

4.1.1 V-UNI-VUL-001: Underconstrained Circuit allows Invalid Comparison

Severity	Critical	**Commit**	0985a28
Type	Underconstrained Circuit	**Status**	Fixed
File(s)	bigComparators.circom		
Location(s)	BigLessThan, UpperLessThan		

Due to restrictions on how constraints must be expressed in ZK circuits, it is often useful to convert signals to their bit representation to perform some computation. To do so, CircomLib provides a template called Num2Bits(N) that converts a signal to an equivalent representation as a bit array of size N. The large prime itself, however, is 254 bits, which allows some values to have multiple representations as a 254 bit array modulo P. For example, 0 can be expressed either as 0 or as P since converting P from a bit array back to a signal will result in 0 due to an overflow. As a result, it is unsafe to use Num2Bits(254) as it is technically unconstrained. Instead, CircomLib provides an alternate template called Num2Bits_strict that performs additional checks to ensure values are not aliased.

```
template BigLessThan() {
    ...

    component bits[2];
    for (var x = 0; x < 2; x++) {
        bits[x] = Num2Bits(254);
        bits[x].in <== in[x];
    }

    ...
}
```

Snippet 4.1: Snippet with an unsafe use of Num2Bits(254)

Impact While Num2Bits(254) is frequently used, most locations contain additional constraints that prevent the potential for aliasing. However, the UpperLessThan and BigLessThan circuits contain unsafe uses of Num2Bits to perform comparisons over signals up to the size of the prime. Additionally, these templates make extensive use out of the binary form produced by Num2Bits. As a result, users can submit proofs of relationships that do not actually hold. For example, a user can improperly prove that 1 < 0 by using the 254 bit representation of P rather than 0.

Recommendation Use Num2Bits_strict rather than Num2Bits(254).

4.1.2 V-UNI-VUL-002: Missing Range Checks on Comparison Circuits

Severity	Critical	**Commit**	`0985a28`
Type	Missing Range Check	**Status**	Fixed
File(s)	`proveReputation.circom`, `userStateTransition.circom`, ...		
Location(s)	ProveReputation, UserStateTransition, EpochKeyLite		

Some computations, such as comparisons, cannot be directly expressed as constraints in Circom. Instead, CircomLib provides templates that developers can use so that they don't have to implement them from scratch. Importantly, some of these templates make assumptions about the inputs that the developers must enforce. The `LessThan(N)`, `LessEqThan(N)`, `GreaterThan(N)` and `GreaterEqThan(N)` templates are all examples of such library templates as they all assume that the inputs are all constrained to be within `N` bits. If this does not hold, the template output can be manipulated. As an example, consider the snippet below. The above code does not include a check that `nonce` is within the range of an 8 bit value. This allows a malicious user to manipulate the circuit by providing a very large `nonce`, say `P - 1`, causing an overflow in the `LessThan` circuit (where `P` is the large prime). This will result in the template improperly reporting that `P - 1` is less than `EPOCH_KEY_NONCE_PER_EPOCH`.

```
template EpochKeyLite(EPOCH_KEY_NONCE_PER_EPOCH) {
    ...

    component nonce_lt = LessThan(8);
    nonce_lt.in[0] <== nonce;
    nonce_lt.in[1] <== EPOCH_KEY_NONCE_PER_EPOCH;
    nonce_lt.out === 1;

    ...
}
```

Snippet 4.2: Snippet of `EpochKeyLite` with an unsafe use of `LessThan`

Impact Several templates make extensive use of the comparator templates from CircomLib, including `ProveReputation`, `UserStateTransition` and `EpochKeyLite`. These circuits implement critical behaviors in the project and can be manipulated in ways the developers likely do not intend. For example, the above snippet is used to ensure that a user can only generate a limited number of keys within an epoch. Without the range check, users can generate about 255 more keys than intended.

Recommendation While a previous auditor recommended that some of these range checks should be removed to save constraints, their removal compromises the security of the system, especially when the inputs are private (as is the case with `nonce` above). Ensure that a range check is performed on all inputs to the comparator circuits.

4.1.3 V-UNI-VUL-003: Potential Overflow in User State Transition

Severity	Medium	**Commit**	`0985a28`
Type	Integer Overflow	**Status**	Intended Behavior
File(s)	`userStateTransition.circom`		
Location(s)	UserStateTransition		

The Unirep protocol defines different types of data that will be combined in different ways when a user transitions their state. One type of data is "Sum Data Fields" that will be combined by summing the user's previous attestations as well as all attestations across epoch keys. There is nothing preventing these data entries from overflowing though, allowing accidental or intentional harm to users via an attester especially since reputation is stored as a sum field.

```
template UserStateTransition(
  STATE_TREE_DEPTH,
  EPOCH_TREE_DEPTH,
  HISTORY_TREE_DEPTH,
  EPOCH_KEY_NONCE_PER_EPOCH,
  FIELD_COUNT,
  SUM_FIELD_COUNT,
  REPL_NONCE_BITS
) {
    ...

      for (var j = 0; j < SUM_FIELD_COUNT; j++) {
        if (i == 0) {
          final_data[i][j] <== data[j] + new_data[i][j];
        } else {
          final_data[i][j] <== final_data[i-1][j] + new_data[i][j];
        }
      }

    ...
}
```

Snippet 4.3: Location where data accumulation may overflow

Impact This depends partially on how attesters make use of these fields. However, since positive and negative reputation are stored in separate data fields, it could allow a user to lose their positive reputation or reset their negative reputation.

Recommendation Consider methods of preventing overflows if they are not desired. Currently an attester might not know if a data value overflows since the specific data values are private to the user. For example, in cases like reputation it may be desirable to specify that a value saturates at the large prime.

Developer Response This behavior is consistent with our documentation. We leave it up to the attester to ensure that values do not overflow (unless intended). For cases like reputation, we make attesters aware of this risk and advise them to use (relatively) small reputation increments so that the likelihood of an overflow occurring is practically non-existent due to the size of the large prime.

4.1.4 V-UNI-VUL-004: Potential Overflow when Proving Reputation

Severity	Medium	**Commit**	0985a28
Type	Integer Overflow	**Status**	Fixed
File(s)	proveReputation.circom		
Location(s)	ProveReputation		

The `ProveReputation` template allows users to prove that their reputation is sufficient to meet certain conditions set by protocols. For example, this circuit allows users to prove that their reputation exceeds some lower-bound or that it is less than some upper-bound. As these checks are performed, however, the circuit has the potential to overflow since there are no checks on the range of `data[0]` or `data[1]`.

```
template ProveReputation(STATE_TREE_DEPTH, EPOCH_KEY_NONCE_PER_EPOCH, SUM_FIELD_COUNT
    , FIELD_COUNT) {
    ...

    component min_rep_check = GreaterEqThan(252);
    min_rep_check.in[0] <== data[0];
    min_rep_check.in[1] <== data[1] + min_rep;

    ...

    component max_rep_check = GreaterEqThan(252);
    max_rep_check.in[0] <== data[1];
    max_rep_check.in[1] <== data[0] + max_rep;

    ...
}
```

Snippet 4.4: Location where arithmetic may overflow in `ProveReputation`

Impact If the “zero reputation” is chosen poorly or if the user has high enough reputation, it is possible for an overflow to allow users to prove that they meet a condition when they do not. Additionally, since it appears that both `data[0]` and `data[1]` are private, such errors could not be detected by the protocol.

Recommendation Perform appropriate range checks on `data[0] + max_rep` and `data[1] + min_rep` to ensure that an overflow does not occur and is constrained to 252 bits as discussed in V-UNI-VUL-002.

4.1.5 V-UNI-VUL-005: Malformed ReusableMerkleTree Root

Severity	Medium	**Commit**	`0985a28`
Type	Logic Error	**Status**	Fixed
File(s)	`ReusableMerkleTree.sol`		
Location(s)	update		

As its name implies, the `ReusableMerkleTree` library allows contracts to create re-usable incremental merkle trees. The content of such trees is stored explicitly in storage and can be "reset" by setting the number of leaves to 0 and changing the root back to the default root. The previous elements in the tree are then overwritten as a new tree is created. As this is a resettable incremental tree, functionality to add to and update the incremental merkle tree is also included. As the merkle tree may contain entries from prior instantiations of the tree, though, care must be taken to ensure to not include stale subtrees. In the update logic, however, this is done by detecting if the latest entry in the subtree is being updated. If it is, additional logic will determine if a zero value should be used when computing the hash. However, using zero value hashes is necessary in more than just the case where the latest entry is updated. As an example, consider a tree with 2 two entries. If the first entry of the tree is updated, it will be hashed together with the second entry when computing the root. However, on the levels above the root computation should hash the current for the subtree with the zero root.

Impact If the tree is updated using the existing logic, at least one hash in the tree will be incorrect until the tree is full. This allows prior entries in the tree to be proven or, in the case where there were no prior entries, allows any path that hashes to 0 at the current level of the tree. While impact of this can be severe, in the case of Unirep, leaves of the tree must change even for specific individuals between resets. As such, for an attacker to take advantage of this, they must essentially compute new data to hash to one of these additional values. As this is extremely difficult to do the likelihood of the attack occurring is low (note, however, that this issue also increase the likelihood of it occurring due to the number of entries that one may collide with).

Recommendation Use the zero value whenever any level of the tree may exceed the current number of leaves. To reduce the likelihood of mistakes, it may be useful to save the zero value in the next entry (if a left entry) when adding to the tree.

```
function update(
    ReusableTreeData storage self,
    uint256 leafIndex,
    uint256 newLeaf
) public returns (uint256) {
    ...

    uint256 hash = newLeaf;
    bool isLatest = leafIndex == leafCount - 1;
    for (uint8 i = 0; i < depth; ) {
        self.elements[indexForElement(i, leafIndex, depth)] = hash;
        uint256[2] memory siblings;
        if (leafIndex & 1 == 0) {
            // it's a left sibling
            siblings = [
                hash,
                isLatest
                    ? defaultZero(i)
                    : self.elements[
                        indexForElement(i, leafIndex + 1, depth)
                    ]
            ];
        } else {
            // it's a right sibling
            uint256 elementIndex = indexForElement(i, leafIndex - 1, depth);
            siblings = [self.elements[elementIndex], hash];
        }

        ...
    }

    ...
}
```

Snippet 4.5: Location in the `update` function where root computation can be incorrect.

4.1.6 V-UNI-VUL-006: Potential Attestation Loss

Severity	Medium	**Commit**	`0985a28`
Type	Logic Error	**Status**	Intended Behavior
File(s)	`Unirep.sol`		
Location(s)	attest		

The Unirep protocol allows attesters to assert that changes should be made to the owners of particular epoch keys. Over the course of the epoch, these changes are recorded the attester's `epochTree` which is a re-usable incremental merkle tree. When epochs expire, users are expected to apply the changes recorded in the `epochTree` by performing a state transition. To do so, however, the epoch tree root must be saved along with the state tree root in the history tree. In the `Unirep` contract, this is done when transitioning to a new epoch but only if an update has been made to the `stateTree`. Therefore, if no state tree update has been made the attestations are discarded.

Impact As the state tree is only changed when a user is added by the attester or when a user transitions state, it is possible for attestations to be lost. In particular, assuming no users are updated it is possible for current users of the protocol to collude to discard undesirable updates to the state (particularly if there are a small number of users). Since this epoch is essentially discarded and ignored, this users may then transition from the previous epoch to the next epoch (i.e. users may skip `e` to transition from `e-1` to `e+1`).

Developer Response It is expected that the attester will validate an epoch key before performing an attestation. As long as this is done properly, then the epoch tree will be empty if the state tree is empty.

```
    function updateEpochIfNeeded(
        uint160 attesterId
    ) public returns (uint48 epoch) {
        ...

        if (attester.stateTree.numberOfLeaves > 0) {
            uint256 historyTreeLeaf = PoseidonT3.hash(
                [attester.stateTree.root, attester.epochTree.root]
            );
            uint256 root = IncrementalBinaryTree.insert(
                attester.historyTree,
                historyTreeLeaf
            );
            attester.historyTreeRoots[root] = true;

            ReusableMerkleTree.reset(attester.stateTree);

            attester.epochTreeRoots[fromEpoch] = attester.epochTree.root;

            emit HistoryTreeLeaf(attesterId, historyTreeLeaf);
        }

        ReusableMerkleTree.reset(attester.epochTree);

        emit EpochEnded(epoch - 1, attesterId);

        attester.currentEpoch = epoch;
    }
```

Snippet 4.6: Location where the `epochTree` is reset

4.1.7 V-UNI-VUL-007: Different Definitions of data[SUM_FIELD_COUNT]

Severity	Medium	**Commit**	0985a28
Type	Logic Error	**Status**	Fixed
File(s)	proveReputation.circom		
Location(s)	ProveReputation		

The Unirep protocol allows attesters to define custom data that will be associated users. This data is defined into two types: sum fields and replacement fields. The data entries between `SUM_FIELD_COUNT` and `FIELD_COUNT` are reserved for replacement data, which reserves the upper bits for an id and the lower bits for the data itself. In the `ProveReputation` circuit, though, the entry at `data[SUM_FIELD_COUNT]` is expected to be a hash as shown below. Note that such a hash will have a value between 0 and the large prime. At other places in the protocol, such as in the `UserStateTransition` circuit, the same data entry is expected to contain replacement data.

```
    component graffiti_hasher = Poseidon(1);
    graffiti_hasher.inputs[0] <== graffiti_pre_image;

    component graffiti_eq = IsEqual();
    graffiti_eq.in[0] <== graffiti_hasher.out;
    graffiti_eq.in[1] <== data[SUM_FIELD_COUNT];
```

Snippet 4.7: Location where `data[SUM_FIELD_COUNT]` is expected to be a hash in the `ProveReputation` circuit

Impact As most locations expect that the data contained at `data[SUM_FIELD_COUNT]` will contain replacement data, it is likely that this function will not work as intended.

```
for (var i = 0; i < EPOCH_KEY_NONCE_PER_EPOCH; i++) {
  // first combine the sum data
  for (var j = 0; j < SUM_FIELD_COUNT; j++) {
    if (i == 0) {
      final_data[i][j] <== data[j] + new_data[i][j];
    } else {
      final_data[i][j] <== final_data[i-1][j] + new_data[i][j];
    }
  }
  // then combine the replacement data
  for (var j = 0; j < REPL_FIELD_COUNT; j++) {
    var field_i = SUM_FIELD_COUNT + j;
    index_check[i][j] = UpperLessThan(REPL_NONCE_BITS);
    index_check[i][j].in[0] <== new_data[i][field_i];
    if (i == 0) {
      index_check[i][j].in[1] <== data[field_i];
    } else {
      index_check[i][j].in[1] <== final_data[i-1][field_i];
    }

    field_select[i][j] = Mux1();
    field_select[i][j].s <== index_check[i][j].out;
    if (i == 0) {
      field_select[i][j].c[1] <== data[field_i];
    } else {
      field_select[i][j].c[1] <== final_data[i-1][field_i];
    }
    field_select[i][j].c[0] <== new_data[i][field_i];

    final_data[i][field_i] <== field_select[i][j].out;
  }
}
```

Snippet 4.8: Location where `data[SUM_FIELD_COUNT]` is expected to be replacement data

4.1.8 V-UNI-VUL-008: Unconstrained Public Input

Severity	Low	**Commit**	`0985a28`
Type	Unconstrained Public Input	**Status**	Fixed
File(s)	`epochKeyLite.circom`		
Location(s)	EpochKeyLite		

The `EpochKeyLite` circuit allows private users to publicly attest values that can be used by applications. Such values are application-specific and do not need circuit validation. If users are not careful, however, it may possible for values be manipulated after the proof is generated. There have been reports of such cases (such as here) however Veridise has been unable to independently verify this attack even on Circom 2.0 with a fixed version of the circuit shown in the issue tracker.

```
template EpochKeyLite(EPOCH_KEY_NONCE_PER_EPOCH) {
    ...

    signal input sig_data;

    ...
}
```

Snippet 4.9: The public input signal that is never used by `EpochKeyLite`

Impact If the report is correct, this allows potentially malicious users to take a valid proof and re-verify it with a different public value. Since this will be used in conjunction with DeFi applications, such unconstrained public signals also provide an opportunity for malicious users to front-run and change public values. This means that DeFi applications cannot prevent these attacks by only allowing a single proof to be submitted.

Recommendation Use a dummy square to constrain `sig_data` as we continue to seek more clarity about this issue.

4.1.9 V-UNI-VUL-009: Replacement Data ID validation not Monotonically Increasing

Severity	Low	**Commit**	`0985a28`
Type	Logic Error	**Status**	Fixed
File(s)	`userStateTransition.circom`		
Location(s)	UserStateTransition		

The Unirep protocol allows attesters to declare “Replacement Data” with content that will be replaced with each attribution. To determine which version of the data should be used, the protocol uses the higher order bits as an ID, where the data with the highest ID should be preserved. In the `UserStateTransition` circuit though, data may be replaced even though the ID does not increase as shown below.

```
  for (var i = 0; i < EPOCH_KEY_NONCE_PER_EPOCH; i++) {
    ...

    // then combine the replacement data
    for (var j = 0; j < REPL_FIELD_COUNT; j++) {
      var field_i = SUM_FIELD_COUNT + j;
      index_check[i][j] = UpperLessThan(REPL_NONCE_BITS);
      index_check[i][j].in[0] <== new_data[i][field_i];
      if (i == 0) {
        index_check[i][j].in[1] <== data[field_i];
      } else {
        index_check[i][j].in[1] <== final_data[i-1][field_i];
      }

      field_select[i][j] = Mux1();
      field_select[i][j].s <== index_check[i][j].out;
      if (i == 0) {
        field_select[i][j].c[1] <== data[field_i];
      } else {
        field_select[i][j].c[1] <== final_data[i-1][field_i];
      }
      field_select[i][j].c[0] <== new_data[i][field_i];

      final_data[i][field_i] <== field_select[i][j].out;
    }
  }
```

Snippet 4.10: Location in UserStateTransition where data may be replaced if it has the same ID

Impact This gives users some choice over the data that they want to be preserved if they can generate data with the same ID. Additionally, if a piece of replacement data has ID 0, it may be accidentally overwritten if a user does not use one of their epoch keys, as in that case the data is required to be 0 (which will have an ID of 0).

Recommendation Enforce that IDs must be monotonically increasing in the circuit.

4.1.10 V-UNI-VUL-010: Manual Signup may be Incorrect

Severity	Low	**Commit**	0985a28
Type	Data Validation	**Status**	Fixed
File(s)	Unirep.sol		
Location(s)	manualUserSignUp		

The Unirep protocol gives attesters the ability to specify unique initial data if desired by using the `manualUserSignUp` function. To do so, the attester must specify the state tree leaf along with initial data associated with the account. The state tree leaf, however, is the hash of several pieces of information, including the initial account data. Currently there is no guarantee, however, that the data provided to `manualUserSignUp` is the same data that was used to calculate the `stateTreeLeaf`.

```
    function manualUserSignUp(
        uint48 epoch,
        uint256 identityCommitment,
        uint256 stateTreeLeaf,
        uint256[] calldata initialData
    ) public {
        _userSignUp(epoch, identityCommitment, stateTreeLeaf);
        if (initialData.length > fieldCount) revert OutOfRange();
        for (uint8 x = 0; x < initialData.length; x++) {
            if (initialData[x] >= SNARK_SCALAR_FIELD) revert InvalidField();
            if (
                x >= sumFieldCount &&
                initialData[x] >= 2 ** (254 - replNonceBits)
            ) revert OutOfRange();
            emit Attestation(
                type(uint48).max,
                identityCommitment,
                uint160(msg.sender),
                x,
                initialData[x]
            );
        }
    }
```

Snippet 4.11: Location where an attester can specify alternate initial data

Impact If the data used to calculate the `stateTreeLeaf` is inconsistent with `initalData`, a user will not be able to use the protocol as knowledge of the data is required to allow a user to properly transition. Therefore, an attester can accidentally or maliciously prevent a user from using their account. Note, that such a user could always sign up with a separate identify commitment but it could be an inconvenience (for example, if there were a reason to use the same `identityCommitment` between applications).

Recommendation While we recognize that the current calculation of the `stateTreeLeaf` would prevent such validation, the leaf calculation could be changed to `h(h(data), h(identity_secret, attester_id, epoch))`.

4.1.11 V-UNI-VUL-011: Replacement Data IDs may not Monotonically Increase

Severity	Low	**Commit**	0985a28
Type	Logic Error	**Status**	Fixed
File(s)	Unirep.sol		
Location(s)	attest		

The Unirep protocol allows attesters to declare "Replacement Data" with content that will be replaced with each attribution. To determine which version of the data should be used, the protocol uses an ID where the data with the highest ID should be preserved. Currently, as shown below, Unirep uses the current timestamp as the replacement data ID. This, however, will not guaranteed that the IDs will increase as any attestations made in the same block (or same transaction if batched) will have the same ID. This allows for ambiguity that could be exploited by users.

```
function attest(
        uint256 epochKey,
        uint48 epoch,
        uint fieldIndex,
        uint change
    ) public {
        ...

        } else {
            if (change >= 2 ** (254 - replNonceBits)) {
                revert OutOfRange();
            }
            change += block.timestamp << (254 - replNonceBits);
            epkData.data[fieldIndex] = change;
        }

        ...
    }
```

Snippet 4.12: Location in the `attest` function where IDs are assigned to replacement data

Impact This gives users some choice over the data that they want to be preserved if they can generate data with the same IDs. As avoiding this would require a slow attestation throughput, it is likely that some replacement data will have the same ID.

Recommendation Use a global counter rather than timestamp.

4.1.12 V-UNI-VUL-012: Private Information Stored in Plaintext

Severity	Low	**Commit**	`0985a28`
Type	Information Leakage	**Status**	Acknowledged
File(s)	`UserState.ts`		
Location(s)	N/A		

Due to the untrusted nature of most client environments such as browsers or phones, it is important to protect important secrets against possible theft. Currently, however, the `UserState` class in Unirep's typescript library stores secret user information as plaintext as it is using Semaphore's `Identity` class as shown below.

```
export default class UserState {
    public id: Identity
    public sync: Synchronizer

    ...
}
```

Snippet 4.13: The `UserState` class with a public Semaphore identity that stores secrets as plaintext

Impact By storing secret information as plaintext, it is possible for malicious applications to steal user secrets.

Recommendation Implement some method of protecting user secrets.

Developer Response We are currently implementing such a feature for the next release of the Unirep protocol.

4.1.13 V-UNI-VUL-013: Unnecessary Num2Bits(254) Constraints

Severity	Warning	**Commit**	0985a28
Type	Unnecessary Constraints	**Status**	Fixed
File(s)	epochKeyLite.circom, modulo.circom, proveReputation.circom		
Location(s)	EpochKeyLite, Modulo, ProveReputation		

Unlike most programming languages, operations in ZK circuits are made over signals that range from 0 to P where P is the large prime. For this reason, it is important to perform range checks using Num2Bits to ensure a signal only accepts values of a particular bit width. The developers have such range check, but rather than checking the bit width using Num2Bits(N) where N is the number of bits, they instead always use Num2Bits(254) and then check that the upper 254 - N bits are 0 as shown below.

```
template Modulo() {
    ...

    // check that remainder and divisor are both < 2**252
    component remainder_bits = Num2Bits(254);
    remainder_bits.in <== remainder;
    for (var x = 252; x < 254; x++) {
        remainder_bits.out[x] === 0;
    }

    ...
}
```

Snippet 4.14: An instance where Num2Bits(254) can be be replaced with Num2Bits(252)

Impact As this pattern is used to perform range checks on specific values, it saves constraints to perform Num2Bits with the desired number of bits.

Recommendation Replace cases that use the above pattern with Num2Bits of the appropriate bit-width.

4.1.14 V-UNI-VUL-014: Add Template Variable Assertions

Severity	Warning	**Commit**	`0985a28`
Type	Data Validation	**Status**	Fixed
File(s)	Multiple		
Location(s)	Multiple		

Circom allows developers to instantiate templates with static values at compile time. This allows templates to declare and uses configurable constants so that they can be easily instantiated and re-used. To prevent unsafe configurations, Circom also allows developers to add assertions over that template parameter assignments must obey. As several Unirep templates make assumptions about the instantiated values of these template parameters, the developers should consider adding assertions over the parameter values.

```
template EpochKeyLite(EPOCH_KEY_NONCE_PER_EPOCH) {
    ...

    component nonce_lt = LessThan(8);
    nonce_lt.in[0] <== nonce;
    nonce_lt.in[1] <== EPOCH_KEY_NONCE_PER_EPOCH;
    nonce_lt.out === 1;

    ...
}
```

Snippet 4.15: An instance where the developers assume `EPOCH_KEY_NONCE_PER_EPOCH < 256`

Impact Configuration errors can be costly, as they could lead to exploits and fixing such errors would require running a new trusted setup ceremony.

Recommendation Such assertions don't add constraints to the circuit and can prevent potential configuration errors. Consider adding appropriate assertions over the template parameters.

4.1.15 V-UNI-VUL-015: Potential for Private Information Leakage

Severity	Warning	**Commit**	`0985a28`
Type	Information Leakage	**Status**	Acknowledged
File(s)	`proveReputation.circom`		
Location(s)	ProveReputation		

ZK circuits partition inputs into those that can be publicly revealed and those that should be kept private as they may contain secret information. In the case of Unirep, a user's overall reputation is intended to be private. However, some of the features allow a user to prove facts about this reputation such as if it is above or below a certain threshold using the `ProveReputation` circuit. Such features necessarily leak information about the private input to other users.

```
template ProveReputation(STATE_TREE_DEPTH, EPOCH_KEY_NONCE_PER_EPOCH, SUM_FIELD_COUNT
    , FIELD_COUNT) {
    ...

    component min_rep_check = GreaterEqThan(252);
    min_rep_check.in[0] <== data[0];
    min_rep_check.in[1] <== data[1] + min_rep;

    component if_not_prove_min_rep = IsZero();
    if_not_prove_min_rep.in <== prove_min_rep;

    component output_rep_check = OR();
    output_rep_check.a <== if_not_prove_min_rep.out;
    output_rep_check.b <== min_rep_check.out;

    output_rep_check.out === 1;

    ...
}
```

Snippet 4.16: The `ProveReputation` circuit which can leak information about a user's reputation

Impact Depending on how the circuit is used, these features could allow a user to accidentally publish their overall reputation (i.e. `data[0]-data[1]`) even in cases where they don't intend to. Depending on the circumstances (such as the length of the attestation history) this could revel a user's epoch key(s) as well.

Recommendation Consider warning users of this possibility so that they are aware of the risks of revealing too much information about their reputation.

4.1.16 V-UNI-VUL-016: Wasted Storage

Severity	Warning	**Commit**	`0985a28`
Type	Storage Optimization	**Status**	Fixed
File(s)	`IUnirep.sol`		
Location(s)	EpochKeyData		

The Unirep protocol allows Attesters to declare data that is unique to individual epoch keys and users. To hold information about this data, the `IUnirep` interface declares the `EpochKeyData` struct which allocates a static array of size 30 to hold the data. The `Unirep` contract, however, uses the immutable variable `fieldCount` as the size of the attester data without comparing it against 30.

```
    struct EpochKeyData {
        uint256 leaf;
        uint256[30] data;
        uint48 leafIndex;
        uint48 epoch;
    }
```

Snippet 4.17: Unnecessarily large `data` struct field

Impact In the event that `fieldCount` is less than 30, storage will be wasted as the `Unirep` contract maintains an `EpochKeyData` entry for every epoch key that is attested to. If `fieldCount` is greater than 30, the contract will be broken.

Recommendation Ideally change the size of `data` to be `fieldCount`. If that is not done, add a requirement that `fieldCount <= 30` in the constructor to prevent potential initialization errors.

4.1.17 V-UNI-VUL-017: Assumed Replacement Data Ordering

Severity	Warning	**Commit**	0985a28
Type	Usability Issue	**Status**	Fixed
File(s)	UserState.ts		
Location(s)	getData		

The Unirep protocol allows attesters to declare “Replacement Data” with content that will be replaced with each attribution as dictated by the data’s id. In Unirep’s typescript library, though, they assume that the latest attribution is the one with the highest id and therefore the one that should be preserved. While the Unirep developers have switched to a global ID that should match this expectation, it does not consider the case where users of the protocol override the id to provide different behaviors.

```
    public getData = async (
        _toEpoch?: number,
        _attesterId: bigint | string = this.sync.attesterId
    ): Promise<bigint[]> => {
        ...

        if (orClauses.length === 0) return data
        const attestations = await this.sync._db.findMany(’Attestation’, {
            where: {
                OR: orClauses,
                attesterId: attesterId,
            },
            orderBy: {
                index: ’asc’,
            },
        })
        for (const a of attestations) {
            const { fieldIndex } = a
            if (fieldIndex < this.sync.settings.sumFieldCount) {
                data[fieldIndex] = (data[fieldIndex] + BigInt(a.change)) % F
            } else {
                data[fieldIndex] = BigInt(a.change)
            }
        }
        return data
    }
```

Snippet 4.18: Location where replacement data is set based on attestation order

Impact As the above function is intended for use by users who will be submitting proofs to the ZK circuits, if the correct data is not fetched, users will be unable to submit valid proofs to the ZK circuits.

Recommendation While we do note that this is consistent with Unirep’s behavior after the switch to a global ID counter, it deviates from the behavior that Unirep has in their documentation. If this library is intended to be used by those who extend Unirep, the developers should consider slightly modifying this logic to be consistent with the documentation they have available for developers.

4.1.18 V-UNI-VUL-018: Confusing Corner Case Logic

Severity	Warning	**Commit**	0985a28
Type	Usability Issue	**Status**	Fixed
File(s)	Synchronizer.ts,UserState.ts		
Location(s)	latestStateTreeLeafIndex, genEpochTree		

While processing on-chain data, it is possible to reach a state that is unexpected by the typescript library. In most cases, the Unirep developers throw an error in this case. In a few isolated locations, such as those shown below, some confusing logic has been added to handle corner cases.

```
    async latestStateTreeLeafIndex(
        _epoch?: number,
        _attesterId: bigint | string = this.sync.attesterId
    ): Promise<number> {
        ...
        if (latestTransitionedEpoch === 0) {
            ...
            if (!signup) {
                throw new Error('@unirep/core:UserState: user is not signed up')
            }
            **if (signup.epoch !== currentEpoch) {
                return 0
            }**
            ...
        }
        ...
    }
```

Snippet 4.19: Location where the zero index is returned in what appears to be an error case

Impact In these cases developers may not correctly understand the value being returned, which could lead to programming errors. In addition, if these checks are solving a specific problem, it should be documented so Unirep developers do not remove the code while refactoring.

Recommendation Add additional documentation about the expected inputs and outputs of the APIs, noting in particular corner cases that the developers handle specially.

```
    async genEpochTree(
        _epoch: number | ethers.BigNumberish,
        attesterId: bigint | string = this.attesterId
    ): Promise<IncrementalMerkleTree> {
        ...
        if (leaves.length === 0) tree.insert(0)
        for (const { hash } of leaves) {
            tree.insert(hash)
        }
        return tree
    }
```

Snippet 4.20: Location where a leaf is inserted if epoch tree is empty

4.1.19 V-UNI-VUL-019: Containerize TypeScript Classes

Severity	Warning	**Commit**	`0985a28`
Type	Access Control	**Status**	Fixed
File(s)	`Synchronizer.ts`, `UserState.ts`		
Location(s)	Synchronizer, UserState		

Unirep provides a typescript library that allows users to interact with the protocol. The library defines a set of classes that collates on-chain data and allows users to easily interact with the application without needing to track the on-chain state themselves. As shown below, several of these classes declare important data as public, which could allow accidental modification by external entities.

```
export class Synchronizer extends EventEmitter {
    public _db: DB
    prover: Prover
    provider: any
    unirepContract: ethers.Contract
    private _attesterId: bigint[] = []
    public settings: any
    private _attesterSettings: { [key: string]: AttesterSetting } = {}
    protected defaultStateTreeLeaf: bigint = BigInt(0)
    protected defaultEpochTreeLeaf: bigint = BigInt(0)
    private _syncAll = false

    private _eventHandlers: any
    private _eventFilters: any

    private pollId: string | null = null
    public pollRate: number = 5000
    public blockRate: number = 100000

    private setupComplete = false

    private lock = new AsyncLock()

    ...
}
```

Snippet 4.21: Synchronizer class defines contract settings as public, allowing accidental modification

Recommendation Similar to other protocols like Semaphore, rather than declaring data as public, containerize the class with getters so that developers cannot accidentally modify crutial settings.

```
export default class UserState {
    public id: Identity
    public sync: Synchronizer

    ...
}
```

Snippet 4.22: UserState class defines private identity as public, allowing accidental modification

Formal Verification 5

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). Note that due to the size and complexity of the proofs, we will not include them in the official report, the circuit definitions and proofs can be found in the following locations:

- Coda Circuits: `https://github.com/Veridise/Coda/tree/certcom/dsl/circuits/unirep`
- Proofs: `https://github.com/Veridise/Coda/tree/certcom/BigInt/src/Benchmarks/Unirep`

Table 5.1 summarizes the specifications and their verification status:

Table 5.1: Summary of Discovered Vulnerabilities.

ID	Description	Status
V-UNI-SPEC-001	MerkleTreeInclusionProof Functional Correctness	Verified
V-UNI-SPEC-002	EpochKeyHasher Functional Correctness	Verified
V-UNI-SPEC-003	EpochTreeLeaf Functional Correctness	Verified
V-UNI-SPEC-004	StateTreeLeaf Functional Correctness	Verified
V-UNI-SPEC-005	IdentitySecret Functional Correctness	Verified
V-UNI-SPEC-006	IdentityCommitment Functional Correctness	Verified
V-UNI-SPEC-007	UpperLessThan Functional Correctness	Verified
V-UNI-SPEC-008	replFieldEqual Functional Correctness	Verified
V-UNI-SPEC-009	Signup Functional Correctness	Verified
V-UNI-SPEC-010	EpochKeyLite Functional Correctness	Verified
V-UNI-SPEC-011	EpochKey Functional Correctness	Verified
V-UNI-SPEC-012	PreventDoubleAction Functional Correctness	Verified
V-UNI-SPEC-013	ProveReputation Functional Correctness	Verified
V-UNI-SPEC-014	UserStateTransition Functional Correctness	Verified

5.1 Detailed Description of Formal Verification Results

5.1.1 V-UNI-SPEC-001: MerkleTreeInclusionProof Functional Correctness

Commit	510c971	**Status**	Verified
Files	incrementalMerkleTree.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.

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

```
let mrkl_tree_incl_pf =
  Circuit
  { name= "MerkleTreeInclusionProof"
  ; inputs=
      [ ("n_levels", tnat)
      ; ("leaf", tf)
      ; ("path_index", tarr_tf n_levels)
      ; ("path_elements", tarr_tf n_levels) ]
  ; outputs= [("root", t_r)]
  ; dep= None
  ; body=
      elet "leaf_zero" (call "IsZero" [leaf])
        (elet "u0"
           (assert_eq (v "leaf_zero") f0)
           (elet "z" (zip path_index path_elements) (hasher z n_levels leaf)) )
  }
```

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

```
let _i = v "_i"
let x = v "x"
let m = v "m"
let c = v "c"
let n_levels = v "n_levels"
let leaf = v "leaf"
let path_index = v "path_index"
let path_elements = v "path_elements"
let z = v "z"
let u_hasher z init = unint "MrklTreeInclPfHash" [z; init]
let u_zip xs ys = unint "zip" [xs; ys]
let z_i_0 z = tget (get z _i) 0
let z_i_1 z = tget (get z _i) 1

let lam_mtip z =
  lama "_i" tint
    (lama "x" tf
       (elet "u0"
          (* path_index[i] binary *)
          (assert_eq (fmul (z_i_0 z) (fsub f1 (z_i_0 z))) f0)
          (elet "c"
             (const_array (tarr_tf z2)
                [const_array tf [x; z_i_1 z]; const_array tf [z_i_1 z; x]] )
             (elet "m"
                (call "MultiMux1" [z2; c; z_i_0 z])
                (call "Poseidon" [z2; m]) ) ) ) )

let hasher z len init =
  iter z0 len (lam_mtip z) ~init ~inv:(fun i ->
      tfq (qeq nu (u_hasher (u_take i z) init)) )

(* {F | nu = #MrklTreeInclPfHash (zip pathIndices siblings) leaf } *)
let t_r =
  tfq
    (qand
       (qeq nu (u_hasher (u_zip path_index path_elements) leaf))
       (qnot (qeq (v "leaf") f0)) )
```

5.1.2 V-UNI-SPEC-002: EpochKeyHasher Functional Correctness

Commit	510c971	**Status**	Verified
Files	leafHasher.circom		
Functions	EpochKeyHasher		

Description The output of the circuit is the poseidon hash of the user's identity secret and a combination of the attester id, epoch and nonce.

Formal Definition The following shows the formal definition for the `EpochKeyHasher` template:

```
let epoch_key_hasher =
  Circuit
    { name= "EpochKeyHasher"
    ; inputs=
        [ ("identity_secret", tf)
        ; ("attester_id", tf)
        ; ("epoch", tf)
        ; ("nonce", tf) ]
    ; outputs=
        [("out", t_epoch_key_hasher identity_secret attester_id epoch nonce)]
    ; dep= None
    ; body=
        call "Poseidon"
          [ z2
          ; const_array tf
              [ identity_secret
              ; fadds
                  [ attester_id
                  ; fmul (fpow f2 (zn 160)) epoch
                  ; fmul (fpow f2 (zn 208)) nonce ] ] ] }
```

Formal Specification The following shows the formal specification for the `EpochKeyHasher` template:

```
let identity_secret = v "identity_secret"
let attester_id = v "attester_id"
let epoch = v "epoch"
let nonce = v "nonce"
(* EpochKeyHasher *)

let t_epoch_key_hasher identity_secret attester_id epoch nonce =
  tfq
    (qeq nu
       (u_poseidon z2
          (const_array tf
             [ identity_secret
             ; fadds
```

```
         [ attester_id
         ; fmul (fpow f2 (zn 160)) epoch
         ; fmul (fpow f2 (zn 208)) nonce ] ] ) ) )
```

5.1.3 V-UNI-SPEC-003: EpochTreeLeaf Functional Correctness

Commit	510c971	**Status**	Verified
Files	leafHasher.circom		
Functions	EpochTreeLeaf		

Description The output of the circuit is the poseidon hash of the user's secret data and the epoch key.

Formal Definition The following shows the formal definition for the `EpochTreeLeaf` template:

```
let epoch_tree_leaf =
  Circuit
    { name= "EpochTreeLeaf"
    ; inputs=
        [("FIELD_COUNT", tnat); ("epoch_key", tf); ("data", tarr_tf field_count)]
    ; outputs= [("out", t_epoch_tree_leaf)]
    ; dep= None
    ; body= iter z0 field_count lam_eptl ~init:(v "epoch_key") ~inv:inv_eptl }
```

Formal Specification The following shows the formal specification for the `EpochTreeLeaf` template:

```
let field_count = v "FIELD_COUNT"

let lam_eptl =
  lama "i" tint
    (lama "x" tf
       (call "Poseidon" [z2; const_array tf [v "x"; get (v "data") (v "i")]]) )

let u_epoch_tree_leaf a b = unint "u_epoch_tree_leaf" [a; b]

let inv_eptl i =
  tfq (qeq nu (u_epoch_tree_leaf (take (v "data") i) (v "epoch_key")))

let t_epoch_tree_leaf =
  tfq (qeq nu (u_epoch_tree_leaf (v "data") (v "epoch_key")))
```

5.1.4 V-UNI-SPEC-004: StateTreeLeaf Functional Correctness

Commit	510c971	**Status**	Verified
Files	leafHasher.circom		
Functions	StateTreeLeaf		

Description The output of the circuit is the poseidon hash of the user's secret data, the user's identity secret, and a combination of the attester id and epoch.

Formal Definition The following shows the formal definition for the `StateTreeLeaf` template:

```
let state_tree_leaf =
  Circuit
    { name= "StateTreeLeaf"
    ; inputs=
        [ ("FIELD_COUNT", tnat)
        ; ("data", tarr_tf field_count)
        ; ("identity_secret", tf)
        ; ("attester_id", tf)
        ; ("epoch", tf) ]
    ; outputs=
        [("out", t_state_tree_leaf identity_secret attester_id epoch (v "data"))]
    ; dep= None
    ; body=
        elet "out1"
          (iter z0 (nsub field_count z1) lam_stl
             ~init:(get (v "data") z0)
             ~inv:inv_stl )
          (call "Poseidon"
             [ z3
             ; const_array tf
                 [ identity_secret
                 ; fadd attester_id (fmul (fpow f2 (zn 160)) epoch)
                 ; v "out1" ] ] ) }
```

Formal Specification The following shows the formal specification for the `StateTreeLeaf` template:

```
let data_drop_1 data = drop data z1

let lam_stl =
  lama "i" tint
    (lama "x" tf
       (call "Poseidon"
          [z2; const_array tf [v "x"; get (data_drop_1 (v "data")) (v "i")]] ) )

let u_state_tree_leaf a b = unint "u_state_tree_leaf" [a; b]

let inv_stl i =
  tfq
```

```
    (qeq nu
       (u_state_tree_leaf (take (data_drop_1 (v "data")) i) (get (v "data") z0)) )

let t_state_tree_leaf identity_secret attester_id epoch data =
  tfq
    (qeq nu
       (u_poseidon z3
          (const_array tf
             [ identity_secret
             ; fadd attester_id (fmul (fpow f2 (zn 160)) epoch)
             ; u_state_tree_leaf (data_drop_1 data) (get data z0) ] ) ) )
```

5.1.5 V-UNI-SPEC-005: IdentitySecret Functional Correctness

Commit	510c971	**Status**	Verified
Files	identity.circom		
Functions	IdentitySecret		

Description The output of the circuit is the poseidon hash of the user's nullifier and trapdoor.

Formal Definition The following shows the formal definition for the `IdentitySecret` template:

```
let identity_secret1 =
  Circuit
    { name= "IdentitySecret"
    ; inputs= [("nullifier", tf); ("trapdoor", tf)]
    ; outputs= [("out", t_identity_secret)]
    ; dep= None
    ; body= call "Poseidon" [z2; const_array tf [nullifier; trapdoor]] }
```

Formal Specification The following shows the formal specification for the `IdentitySecret` template:

```
let nullifier = v "nullifier"
let trapdoor = v "trapdoor"

let t_identity_secret =
  tfq (qeq nu (u_poseidon z2 (const_array tf [nullifier; trapdoor])))
```

5.1.6 V-UNI-SPEC-006: IdentityCommitment Functional Correctness

Commit	`510c971`	**Status**	Verified
Files	`identity.circom`		
Functions	IdentityCommitment		

Description The output of the circuit is the poseidon hash of the user's identity secret.

Formal Definition The following shows the formal definition for the `IdentityCommitment` template:

```
let identity_commitment =
  Circuit
    { name= "IdentityCommitment"
    ; inputs= [("nullifier", tf); ("trapdoor", tf)]
    ; outputs=
        [ ("secret", t_identity_commitment_secret nullifier trapdoor)
        ; ("out", t_identity_commitment_out nullifier trapdoor) ]
    ; dep= None
    ; body=
        make
          [ call "IdentitySecret" [nullifier; trapdoor]
          ; call "Poseidon"
              [ z1
              ; const_array tf
                  [u_poseidon z2 (const_array tf [nullifier; trapdoor])] ] ] }
```

Formal Specification The following shows the formal specification for the `IdentityCommitment` template:

```
let t_identity_commitment_out nullifier trapdoor =
  tfq
    (qeq nu
       (u_poseidon z1
          (const_array tf
             [u_poseidon z2 (const_array tf [nullifier; trapdoor])] ) ) )

let t_identity_commitment_secret nullifier trapdoor =
  tfq (qeq nu (u_poseidon z2 (const_array tf [nullifier; trapdoor])))
```

5.1.7 V-UNI-SPEC-007: UpperLessThan Functional Correctness

Commit	510c971	**Status**	Verified
Files	bigComparators.circom		
Functions	UpperLessThan		

Description The circuit determines whether the upper N bits of the first input is less than the upper N bits of the second input and outputs the result of the comparison.

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

```
let upper_less_than =
  Circuit
    { name= "UpperLessThan"
    ; inputs= [("n", t_n); ("in_", tarr_t_k tf z2)]
    ; outputs= [("out", t_upper_less_than_out)]
    ; dep= None
    ; body=
        elet "bits_0"
          (call "Num2Bits" [zn 254; get (v "in_") (zn 0)])
          (elet "bits_1"
             (call "Num2Bits" [zn 254; get (v "in_") (zn 1)])
             (elet "alias0"
                (call "AliasCheck" [v "bits_0"])
                (elet "alias1"
                   (call "AliasCheck" [v "bits_1"])
                   (elet "upper_bits_0"
                      (call "Bits2Num"
                         [v "n"; drop (v "bits_0") (nsub (zn 254) (v "n"))] )
                      (elet "upper_bits_1"
                         (call "Bits2Num"
                            [v "n"; drop (v "bits_1") (nsub (zn 254) (v "n"))] )
                         (elet "lt"
                            (call "LessThan"
                               [v "n"; v "upper_bits_0"; v "upper_bits_1"] )
                            (v "lt") ) ) ) ) ) ) }
```

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

```
let t_n =
  TRef
    ( tint
    , QAnd
        ( lift (leq z0 nu)
        , qand (lift (nu <=. zn 254)) (lift (zn 254 <=. zsub1 CPLen)) ) )

let t_upper_less_than_out =
  tfq
    (ind_dec nu
       (lt
          (zdiv (toUZ (get (v "in_") (zn 0))) (zpow z2 (nsub (zn 254) (v "n"))))
          (zdiv (toUZ (get (v "in_") (zn 1))) (zpow z2 (nsub (zn 254) (v "n")))) ) )
```

5.1.8 V-UNI-SPEC-008: replFieldEqual Functional Correctness

Commit	510c971	Status	Verified
Files	bigComparators.circom		
Functions	replFieldEqual		

Description The circuit determines if the lower N bits of the first input is equal to the lower N bits of the second input and outputs the result of the comparison.

Formal Definition The following shows the formal definition for the `replFieldEqual` template:

```
let repl_field_equal =
  Circuit
    { name= "ReplFieldEqual"
    ; inputs= [("REPL_NONCE_BITS", t_n); ("in_", tarr_t_k tf z2)]
    ; outputs= [("out", t_repl_field_equal_out)]
    ; dep= None
    ; body=
        elet "bits_0"
          (call "Num2Bits" [zn 254; get (v "in_") (zn 0)])
          (elet "bits_1"
             (call "Num2Bits" [zn 254; get (v "in_") (zn 1)])
             (elet "alias0"
                (call "AliasCheck" [v "bits_0"])
                (elet "alias1"
                   (call "AliasCheck" [v "bits_1"])
                   (elet "repl_bits_0"
                      (call "Bits2Num"
                         [ nsub (zn 254) (v "REPL_NONCE_BITS")
                         ; take (v "bits_0")
                             (nsub (zn 254) (v "REPL_NONCE_BITS")) ] )
                      (elet "repl_bits_1"
                         (call "Bits2Num"
                            [ nsub (zn 254) (v "REPL_NONCE_BITS")
                            ; take (v "bits_1")
                                (nsub (zn 254) (v "REPL_NONCE_BITS")) ] )
                         (elet "eq"
                            (call "IsEqual" [v "repl_bits_0"; v "repl_bits_1"])
                            (v "eq") ) ) ) ) ) ) }
```

Formal Specification The following shows the formal specification for the `replFieldEqual` template:

```
let t_n =
  TRef
    ( tint
    , QAnd
        ( lift (leq z0 nu)
        , qand (lift (nu <=. zn 254)) (lift (zn 254 <=. zsub1 CPLen)) ) )

let t_repl_field_equal_out =
  tfq
    (ind_dec nu
       (eq
          (zmod
             (toUZ (get (v "in_") (zn 0)))
             (zpow z2 (nsub (zn 254) (v "REPL_NONCE_BITS"))) )
          (zmod
             (toUZ (get (v "in_") (zn 1)))
             (zpow z2 (nsub (zn 254) (v "REPL_NONCE_BITS"))) ) ) )
```

5.1.9 V-UNI-SPEC-009: Signup Functional Correctness

Commit	510c971	**Status**	Verified
Files	signup.circom		
Functions	Signup		

Description The circuit computes the user's identity commitment and the initial state tree leaf for the user where all data is 0. Both the identity commitment and the state tree leaf are returned.

Formal Definition The following shows the formal definition for the `Signup` template:

```
let signup =
  Circuit
    { name= "Signup"
    ; inputs=
        [ ("FIELD_COUNT", tnat)
        ; ("attester_id", tf)
        ; ("epoch", tf)
        ; ("identity_nullifier", tf)
        ; ("identity_trapdoor", tf) ]
    ; outputs=
        [ ( "identity_commitment"
          , t_identity_commitment_out identity_nullifier identity_trapdoor )
        ; ( "state_tree_leaf"
          , t_state_tree_leaf
              (u_poseidon z2
                 (const_array tf [identity_nullifier; identity_trapdoor]) )
              attester_id epoch (v "all_0") ) ]
    ; dep= None
    ; body=
        elet "all_0"
          (consts_n (v "FIELD_COUNT") f0)
          (match_with’ ["ic_secret"; "ic_out"]
             (call "IdentityCommitment" [identity_nullifier; identity_trapdoor])
             (make
                [ v "ic_out"
                ; call "StateTreeLeaf"
                    [ v "FIELD_COUNT"
                    ; v "all_0"
                    ; v "ic_secret"
                    ; v "attester_id"
                    ; v "epoch" ] ] ) ) }
```

Formal Specification The following shows the formal specification for the `Signup` template:

```
let identity_nullifier = v "identity_nullifier"
let identity_trapdoor = v "identity_trapdoor"
let identity_secret = v "identity_secret"
let reveal_nonce = v "reveal_nonce"
let attester_id = v "attester_id"
let epoch = v "epoch"
let nonce = v "nonce"
let u_state_tree_leaf a b = unint "u_state_tree_leaf" [a; b]
let data_drop_1 data = drop data z1

let t_state_tree_leaf identity_secret attester_id epoch data =
  tfq
    (qeq nu
       (u_poseidon z3
          (const_array tf
             [ identity_secret
             ; fadd attester_id (fmul (fpow f2 (zn 160)) epoch)
             ; u_state_tree_leaf (data_drop_1 data) (get data z0) ] ) ) )

let t_identity_commitment_out nullifier trapdoor =
  tfq
    (qeq nu
       (u_poseidon z1
          (const_array tf
             [u_poseidon z2 (const_array tf [nullifier; trapdoor])] ) ) )
```

5.1.10 V-UNI-SPEC-010: EpochKeyLite Functional Correctness

Commit	510c971	**Status**	Verified
Files	epochKeyLite.circom		
Functions	EpochKeyLite		

Description The circuit computes a user's epoch key leaf as defined by the EpochKeyHasher with the corresponding nonce. Additionally, the circuit will reveal the attester id, epoch and, optionally, the nonce used to calculate the epoch key.

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

```
let epoch_key_lite =
  Circuit
    { name= "EpochKeyLite"
    ; inputs=
        [ ("FIELD_COUNT", tnat)
        ; ( "EPOCH_KEY_NONCE_PER_EPOCH"
          , tnat_e (leq nu (zsub (zpow (zn 2) (zn 8)) z1)) )
        ; ("identity_secret", tf)
        ; ("reveal_nonce", tf)
        ; ("attester_id", tf)
        ; ("epoch", tf)
        ; ("nonce", tf) ]
    ; outputs=
        [ ("control", t_control reveal_nonce attester_id epoch nonce)
        ; ( "epoch_key"
          , t_epoch_key_hasher_out identity_secret attester_id epoch nonce ) ]
    ; dep= None
    ; body=
        elet "reveal_nonce_check"
          (assert_eq (fmul reveal_nonce (fsub reveal_nonce f1)) f0)
          (elet "attester_id_check"
             (call "Num2Bits" [zn 160; v "attester_id"])
             (elet "epoch_bits"
                (call "Num2Bits" [zn 48; v "epoch"])
                (elet "nonce_range_check"
                   (call "Num2Bits" [zn 8; v "nonce"])
                   (elet "nonce_lt"
                      (call "LessThan"
                         [zn 8; v "nonce"; nat2f (v "EPOCH_KEY_NONCE_PER_EPOCH")] )
                      (elet "u0"
                         (assert_eq (v "nonce_lt") f1)
                         (elet "ctrl"
                            (fadds
                               [ fmul reveal_nonce (fpow f2 (zn 232))
                               ; fmul attester_id (fpow f2 (zn 72))
                               ; fmul epoch (fpow f2 (zn 8))
                               ; fmul reveal_nonce nonce ] )
                            (make
                               [ v "ctrl"
                               ; call "EpochKeyHasher"
                                   [ v "identity_secret"
                                   ; v "attester_id"
                                   ; v "epoch"
                                   ; v "nonce" ] ] ) ) ) ) ) ) ) }
```

Formal Specification The following shows the formal specification for the `EpochKeyLite` template:

```
let identity_secret = v "identity_secret"
let reveal_nonce = v "reveal_nonce"
let attester_id = v "attester_id"
let epoch = v "epoch"
let nonce = v "nonce"

let t_epoch_key_hasher identity_secret attester_id epoch nonce =
  tfq
    (qeq nu
       (u_poseidon z2
          (const_array tf
             [ identity_secret
             ; fadds
                 [ attester_id
                 ; fmul (fpow f2 (zn 160)) epoch
                 ; fmul (fpow f2 (zn 208)) nonce ] ] ) ) )

let t_epoch_key_hasher_out identity_secret attester_id epoch nonce =
  t_epoch_key_hasher identity_secret attester_id epoch nonce

let t_control reveal_nonce attester_id epoch nonce =
  tfq
    (qeq nu
       (fadds
          [ fmul reveal_nonce (fpow f2 (zn 232))
          ; fmul attester_id (fpow f2 (zn 72))
          ; fmul epoch (fpow f2 (zn 8))
          ; fmul reveal_nonce nonce ] ) )
```

5.1.11 V-UNI-SPEC-011: EpochKey Functional Correctness

Commit	`510c971`	**Status**	Verified
Files	`epochKey.circom`		
Functions	EpochKey		

Description The circuit computes a user's epoch key leaf as defined by the `EpochKeyHasher` with the corresponding nonce. Additionally, the circuit will reveal the attester id, epoch and, optionally, the nonce used to calculate the epoch key. Finally, the circuit computes the user's state tree leaf with the input data and calculates the root of the merkle tree with the corresponding leaf and siblings.

Formal Definition The following shows the formal definition for the `EpochKey` template:

```
let epoch_key =
  Circuit
    { name= "EpochKey"
    ; inputs=
        [ ("STATE_TREE_DEPTH", t_n)
        ; ("EPOCH_KEY_NONCE_PER_EPOCH", t_n)
        ; ("FIELD_COUNT", tnat)
        ; ("state_tree_indexes", tarr_t_k tf (v "STATE_TREE_DEPTH"))
        ; ("state_tree_elements", tarr_t_k tf (v "STATE_TREE_DEPTH"))
        ; ("identity_secret", tf)
        ; ("reveal_nonce", tf)
        ; ("attester_id", tf)
        ; ("epoch", tf)
        ; ("nonce", tf)
        ; ("data", tarr_t_k tf (v "FIELD_COUNT"))
        ; ("sig_data", tf) ]
    ; outputs=
        [ ( "epoch_key"
          , t_epoch_key_hasher_out identity_secret attester_id epoch nonce )
        ; ( "state_tree_root"
          , t_r (v "state_tree_indexes") (v "state_tree_elements")
              identity_secret attester_id epoch (v "data") )
        ; ("control", t_control reveal_nonce attester_id epoch nonce) ]
    ; dep= None
    ; body=
        elet "leaf_hasher"
          (call "StateTreeLeaf"
             [ v "FIELD_COUNT"
             ; v "data"
             ; v "identity_secret"
             ; v "attester_id"
             ; v "epoch" ] )
          (elet "merkletree"
             (call "MerkleTreeInclusionProof"
                [ v "STATE_TREE_DEPTH"
                ; v "leaf_hasher"
                ; v "state_tree_indexes"
                ; v "state_tree_elements" ] )
             (match_with' ["control"; "epoch_key"]
                (call "EpochKeyLite"
                   [ v "FIELD_COUNT"
                   ; v "EPOCH_KEY_NONCE_PER_EPOCH"
                   ; v "identity_secret"
                   ; v "reveal_nonce"
                   ; v "attester_id"
                   ; v "epoch"
                   ; v "nonce" ] )
                (make [v "epoch_key"; v "merkletree"; v "control"]) ) ) }
```

Formal Specification The following shows the formal specification for the `EpochKey` template:

```
let identity_secret = v "identity_secret"
let reveal_nonce = v "reveal_nonce"
let attester_id = v "attester_id"
let epoch = v "epoch"
let nonce = v "nonce"
let u_hasher z init = unint "MrklTreeInclPfHash" [z; init]
let u_zip xs ys = unint "zip" [xs; ys]
let u_state_tree_leaf a b = unint "u_state_tree_leaf" [a; b]
let data_drop_1 data = drop data z1

let t_epoch_key_hasher identity_secret attester_id epoch nonce =
  tfq
    (qeq nu
       (u_poseidon z2
          (const_array tf
             [ identity_secret
             ; fadds
                 [ attester_id
                 ; fmul (fpow f2 (zn 160)) epoch
                 ; fmul (fpow f2 (zn 208)) nonce ] ] ) ) )

let t_r path_index path_elements identity_secret attester_id epoch data =
  tfq
    (qeq nu
       (u_hasher
          (u_zip path_index path_elements)
          (u_poseidon z3
             (const_array tf
                [ identity_secret
                ; fadd attester_id (fmul (fpow f2 (zn 160)) epoch)
                ; u_state_tree_leaf (data_drop_1 data) (get data z0) ] ) ) ) )

let t_epoch_key_hasher_out identity_secret attester_id epoch nonce =
  t_epoch_key_hasher identity_secret attester_id epoch nonce

let t_control reveal_nonce attester_id epoch nonce =
  tfq
    (qeq nu
       (fadds
          [ fmul reveal_nonce (fpow f2 (zn 232))
          ; fmul attester_id (fpow f2 (zn 72))
          ; fmul epoch (fpow f2 (zn 8))
          ; fmul reveal_nonce nonce ] ) )

let t_n =
  TRef
    ( tint
    , QAnd
        ( lift (leq z0 nu)
        , qand (lift (nu <=. zn 254)) (lift (zn 254 <=. zsub1 CPLen)) ) )
```

5.1.12 V-UNI-SPEC-012: PreventDoubleAction Functional Correctness

Commit	510c971	**Status**	Verified
Files	preventDoubleAction.circom		
Functions	PreventDoubleAction		

Description The circuit will essentially an epoch key proof as defined in the `EpochKey` template. Additionally, it will compute a nullifier as the poseidon hash of the user's identity nullifier and external nullifier. Finally, it will compute a user's identity commitment.

Formal Definition The following shows the formal definition for the `PreventDoubleAction` template:

```
let prevent_double_action =
  Circuit
    { name= "PreventDoubleAction"
    ; inputs=
        [ ("STATE_TREE_DEPTH", t_n)
        ; ("EPOCH_KEY_NONCE_PER_EPOCH", t_n)
        ; ("FIELD_COUNT", tnat)
        ; ("state_tree_indexes", tarr_t_k tf (v "STATE_TREE_DEPTH"))
        ; ("state_tree_elements", tarr_t_k tf (v "STATE_TREE_DEPTH"))
        ; ("reveal_nonce", tf)
        ; ("attester_id", tf)
        ; ("epoch", tf)
        ; ("nonce", tf)
        ; ("sig_data", tf)
        ; ("identity_nullifier", tf)
        ; ("external_nullifier", tf)
        ; ("identity_trapdoor", tf)
        ; ("data", tarr_t_k tf (v "FIELD_COUNT")) ]
    ; outputs=
        [ ( "epoch_key"
          , t_epoch_key_hasher_out identity_secret attester_id epoch nonce )
        ; ( "state_tree_root"
          , t_r (v "state_tree_indexes") (v "state_tree_elements")
              identity_secret attester_id epoch (v "data") )
        ; ( "nullifier"
          , tfq
              (qeq nu
                 (u_poseidon z2
                    (const_array tf
                       [v "identity_nullifier"; v "external_nullifier"] ) ) ) )
        ; ( "identity_commitment"
          , t_identity_commitment_out (v "identity_nullifier")
              (v "identity_trapdoor") )
        ; ( "control"
          , t_control (v "reveal_nonce") (v "attester_id") (v "epoch")
              (v "nonce") ) ]
    ; dep= None
    ; body=
        elet "nullifier"
          (call "Poseidon"
             [ zn 2
             ; const_array tf [v "identity_nullifier"; v "external_nullifier"]
             ] )
          (match_with' ["identity_secret"; "out"]
             (call "IdentityCommitment"
                [v "identity_nullifier"; v "identity_trapdoor"] )
             (elet "leaf_hasher"
                (call "StateTreeLeaf"
                   [ v "FIELD_COUNT"
                   ; v "data"
                   ; v "identity_secret"
                   ; v "attester_id"
                   ; v "epoch" ] )
                (elet "merkletree"
                   (call "MerkleTreeInclusionProof"
                      [ v "STATE_TREE_DEPTH"
                      ; v "leaf_hasher"
                      ; v "state_tree_indexes"
                      ; v "state_tree_elements" ] )
                   (match_with' ["control"; "epoch_key"]
                      (call "EpochKeyLite"
                         [ v "FIELD_COUNT"
                         ; v "EPOCH_KEY_NONCE_PER_EPOCH"
                         ; v "identity_secret"
                         ; v "reveal_nonce"
                         ; v "attester_id"
                         ; v "epoch"
                         ; v "nonce" ] )
                      (make
                         [ v "epoch_key"
                         ; v "merkletree"
                         ; v "nullifier"
                         ; v "out"
                         ; v "control" ] ) ) ) ) ) ) }
```

Formal Specification The following shows the formal specification for the `PreventDoubleAction` template:

```
let identity_secret = v "identity_secret"
let reveal_nonce = v "reveal_nonce"
let attester_id = v "attester_id"
let epoch = v "epoch"
let nonce = v "nonce"

let t_identity_commitment_out nullifier trapdoor =
  tfq
    (qeq nu
       (u_poseidon z1
          (const_array tf
             [u_poseidon z2 (const_array tf [nullifier; trapdoor])] ) ) )
```

```

let t_control reveal_nonce attester_id epoch nonce =
  tfq
    (qeq nu
       (fadds
          [ fmul reveal_nonce (fpow f2 (zn 232))
          ; fmul attester_id (fpow f2 (zn 72))
          ; fmul epoch (fpow f2 (zn 8))
          ; fmul reveal_nonce nonce ] ) )

let u_hasher z init = unint "MrklTreeInclPfHash" [z; init]

let u_zip xs ys = unint "zip" [xs; ys]

let t_n =
  TRef
    ( tint
    , QAnd
        ( lift (leq z0 nu)
        , qand (lift (nu <=. zn 254)) (lift (zn 254 <=. zsub1 CPLen)) ) )

let t_epoch_key_hasher identity_secret attester_id epoch nonce =
  tfq
    (qeq nu
       (u_poseidon z2
          (const_array tf
             [ identity_secret
             ; fadds
                 [ attester_id
                 ; fmul (fpow f2 (zn 160)) epoch
                 ; fmul (fpow f2 (zn 208)) nonce ] ] ) ) ) )

let t_epoch_key_hasher_out identity_secret attester_id epoch nonce =
  t_epoch_key_hasher identity_secret attester_id epoch nonce

let u_state_tree_leaf a b = unint "u_state_tree_leaf" [a; b]

let data_drop_1 data = drop data z1

let t_r path_index path_elements identity_secret attester_id epoch data =
  tfq
    (qeq nu
       (u_hasher
          (u_zip path_index path_elements)
          (u_poseidon z3
             (const_array tf
                [ identity_secret
                ; fadd attester_id (fmul (fpow f2 (zn 160)) epoch)
                ; u_state_tree_leaf (data_drop_1 data) (get data z0) ] ) ) ) ) )
```

5.1.13 V-UNI-SPEC-013: ProveReputation Functional Correctness

Commit	510c971	Status	Verified
Files	proveReputation.circom		
Functions	ProveReputation		

Description The circuit will compute an epoch key proof according to the EpochKey template. It will also optionally prove information about a user's reputation, including that it above some threshold, below some threshold or is zero. Finally, the circuit will optionally prove that the user's *graffiti* data value is equal to an input value.

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

```
let prove_reputation =
  Circuit
    { name= "ProveReputation"
    ; inputs=
      [ ("STATE_TREE_DEPTH", t_n)
      ; ("EPOCH_KEY_NONCE_PER_EPOCH", t_n)
      ; ("SUM_FIELD_COUNT", tnat_e (nu <. v "FIELD_COUNT"))
      ; ("FIELD_COUNT", tnat)
      ; ("REPL_NONCE_BITS", t_n)
      ; ("identity_secret", tf)
      ; ("state_tree_indexes", tarr_t_k tf (v "STATE_TREE_DEPTH"))
      ; ("state_tree_elements", tarr_t_k tf (v "STATE_TREE_DEPTH"))
      ; ("data", tarr_t_k tf (v "FIELD_COUNT"))
      ; ("prove_graffiti", tf)
      ; ("graffiti", tf)
      ; ("reveal_nonce", tf)
      ; ("attester_id", tf)
      ; ("epoch", tf)
      ; ("nonce", tf)
      ; ("min_rep", tf)
      ; ("max_rep", tf)
      ; ("prove_min_rep", tf)
      ; ("prove_max_rep", tf)
      ; ("prove_zero_rep", tf)
      ; ("sig_data", tf) ]
    ; outputs=
      [ ( "epoch_key"
        , t_epoch_key_hasher_out identity_secret attester_id epoch nonce )
      ; ( "state_tree_root"
        , t_r (v "state_tree_indexes") (v "state_tree_elements")
          identity_secret attester_id epoch (v "data") )
      ; ("control", tarr_t_q_k tf u_control z2) ]
    ; dep= Some u_prove_reputation
    ; body=
      elet "min_rep_bits"
        (call "Num2Bits" [zn 64; v "min_rep"])
        (elet "max_rep_bits"
          (call "Num2Bits" [zn 64; v "max_rep"])
          (elet "u0"
            (assert_eq
              (fmul (v "prove_graffiti") (fsub (v "prove_graffiti") f1))
              f0 )
            (elet "u1"
              (assert_eq
                (fmul (v "prove_min_rep") (fsub (v "prove_min_rep") f1))
                f0 )
              (elet "u2"
                (assert_eq
                  (fmul (v "prove_max_rep")
                    (fsub (v "prove_max_rep") f1) )
                  f0 )
              (elet "u3"
                (assert_eq
                  (fmul (v "prove_zero_rep")
                    (fsub (v "prove_zero_rep") f1) )
                  f0 )
                  (elet "control_1"
                    (fadds
                      [ fmuls [v "prove_graffiti"; fpow (fn 2) (zn 131)]
                      ; fmuls [v "prove_zero_rep"; fpow (fn 2) (zn 130)]
                      ; fmuls [v "prove_max_rep"; fpow (fn 2) (zn 129)]
                      ; fmuls [v "prove_min_rep"; fpow (fn 2) (zn 128)]
                      ; fmuls [v "max_rep"; fpow (fn 2) (zn 64)]
                      ; v "min_rep" ] )
                    (elet "epoch_range_check"
                      (call "Num2Bits" [zn 48; v "epoch"])
                      (elet "attester_id_check"
                        (call "Num2Bits" [zn 160; v "attester_id"])
                        (elet "epoch_key_gen"
                          (call "EpochKey"
                            [ v "STATE_TREE_DEPTH"
                            ; v "EPOCH_KEY_NONCE_PER_EPOCH"
                            ; v "FIELD_COUNT"
                            ; v "state_tree_indexes"
                            ; v "state_tree_elements"
                            ; v "identity_secret"
                            ; v "reveal_nonce"
                            ; v "attester_id"
                            ; v "epoch"
                            ; v "nonce"
                            ; v "data"
                            ; v "sig_data" ] )
                          (elet "epoch_key"
                            (tget (v "epoch_key_gen") 0)
                            (elet "state_tree_root"
                              (tget (v "epoch_key_gen") 1)
                              (elet "control_0"
                                (tget (v "epoch_key_gen") 2)
                                (elet "data_0_check"
                 (call "Num2Bits"
                  [zn 64; get (v "data") z0] )
                 (elet "data_1_check"
                  (call "Num2Bits"
                   [zn 64; get (v "data") z1] )
                  (elet "min_rep_check"
                   (call "GreaterEqThan"
                    [ zn 66
                    ; get (v "data") z0
                    ; fadd
                     (get (v "data") z1)
                     (v "min_rep") ] )
                   (elet "if_not_prove_min_rep"
                    (call "IsZero"
                     [v "prove_min_rep"] )
                    (elet "output_rep_check"
                     (call "Or"
                      [ v "if_not_prove_min_rep"
                      ; v "min_rep_check"
                      ] )
                     (elet "u4"
                      (assert_eq
                       (v "output_rep_check" )
                       f1 )
                      (elet "max_rep_check"
                       (call "GreaterEqThan"
                        [ zn 66
                        ; get
                         (v "data" )
                         z1
                        ; fadd
                         (get
                          (v "data" )
                          z0 )
                         (v "max_rep" )
                        ] )
                       (elet "if_not_prove_max_rep"
                        (call "IsZero"
                         [ v "prove_max_rep"
                         ] )
                        (elet "max_rep_check_out"
                         (call "Or"
                          [ v "if_not_prove_max_rep"
                          ; v "max_rep_check"
                          ] )
                        (elet "u5"
                         (assert_eq
                          (v "max_rep_check_out" )
                          f1 )
                         (elet "zero_rep_check"
                          (call "IsEqual"
                           [ get
                           (v "data" )
```

```
                                                  z0
                                                  ; get
                                                  ( v "data" )
                                                  z1
                                                  ] )
                                                (elet "if_not_prove_zero_rep"
                                                  (call "IsZero"
                                                    [ v "prove_zero_rep"
                                                    ] )
                                                (elet "zero_rep_check_out"
                                                  (call "Or"
                                                    [v "if_not_prove_zero_rep"
                                                    ; v "zero_rep_check"
                                                    ] )
                                                  (elet "u6"
                                                    (assert_eq
                                                      (v "zero_rep_check_out" )
                                                      f1 )
                                                    (elet "if_not_check_graffiti"
                                                      (call "IsZero"
                                                        [v "prove_graffiti"
                                                        ] )
                                                      (elet "repl_field_equal"
                                                        (call "ReplFieldEqual"
                                                          [v "REPL_NONCE_BITS"
                                                          ; cons
                                                          (v "graffiti" )
                                                          (cons
                                                            (get
                                                              (v "data" )
                                                              (v "SUM_FIELD_COUNT
" ) )
                                                            cnil )
                                                          ] )
                                                        (elet "check_graffiti"
                                                          (call "Or"
                                                            [v "
if_not_check_graffiti"
                                                            ; v "repl_field_equal
"
                                                            ] )
                                                          (elet "u7"
                                                            (assert_eq
                                                              (v "check_graffiti"
 )
                                                              f1 )
                                                            (make
                                                              [v "epoch_key"
                                                              ; v "
state_tree_root"
                                                              ; cons
                                                              (v "control_0" )
                                                              (cons
                                                                    (v "control_1" )
                                                                    cnil )
                                                                  ] ) ) ) ) ) ) ) ) )
     ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) ) )
    }
```

Formal Specification The following shows the formal specification for the `ProveReputation` template:

```
let identity_secret = v "identity_secret"
let reveal_nonce = v "reveal_nonce"
let attester_id = v "attester_id"
let epoch = v "epoch"
let nonce = v "nonce"

let t_n =
  TRef
    ( tint
    , QAnd
        ( lift (leq z0 nu)
        , qand (lift (nu <=. zn 254)) (lift (zn 254 <=. zsub1 CPLen)) ) )

let u_state_tree_leaf a b = unint "u_state_tree_leaf" [a; b]

let data_drop_1 data = drop data z1

let u_prove_reputation =
  ands
    [ lift (is_binary (v "prove_graffiti"))
    ; lift (is_binary (v "prove_min_rep"))
    ; lift (is_binary (v "prove_max_rep"))
    ; lift (is_binary (v "prove_zero_rep"))
    ; lift (toUZ (v "min_rep") <. zpow (zn 2) (zn 64))
    ; lift (toUZ (v "max_rep") <. zpow (zn 2) (zn 64))
    ; lift (toUZ (v "epoch") <. zpow (zn 2) (zn 48))
    ; lift (toUZ (v "attester_id") <. zpow (zn 2) (zn 160)) ]

let u_hasher z init = unint "MrklTreeInclPfHash" [z; init]

let u_zip xs ys = unint "zip" [xs; ys]

let u_control =
  ands
    [ qeq (get nu z0)
        (fadds
           [ fmul reveal_nonce (fpow f2 (zn 232))
           ; fmul attester_id (fpow f2 (zn 72))
           ; fmul epoch (fpow f2 (zn 8))
           ; fmul reveal_nonce nonce ] )
    ; qeq (get nu z1)
        (fadds
           [ fmuls [v "prove_graffiti"; fpow (fn 2) (zn 131)]
           ; fmuls [v "prove_zero_rep"; fpow (fn 2) (zn 130)]
           ; fmuls [v "prove_max_rep"; fpow (fn 2) (zn 129)]
           ; fmuls [v "prove_min_rep"; fpow (fn 2) (zn 128)]
           ; fmuls [v "max_rep"; fpow (fn 2) (zn 64)]
           ; v "min_rep" ] ) ]

let t_epoch_key_hasher identity_secret attester_id epoch nonce =
  tfq
    (qeq nu
       (u_poseidon z2
          (const_array tf
             [ identity_secret
             ; fadds
                 [ attester_id
                 ; fmul (fpow f2 (zn 160)) epoch
                 ; fmul (fpow f2 (zn 208)) nonce ] ] ) ) )

let t_r path_index path_elements identity_secret attester_id epoch data =
  tfq
    (qeq nu
       (u_hasher
          (u_zip path_index path_elements)
          (u_poseidon z3
             (const_array tf
                [ identity_secret
                ; fadd attester_id (fmul (fpow f2 (zn 160)) epoch)
                ; u_state_tree_leaf (data_drop_1 data) (get data z0) ] ) ) ) )

let t_epoch_key_hasher_out identity_secret attester_id epoch nonce =
  t_epoch_key_hasher identity_secret attester_id epoch nonce
```

5.1.14 V-UNI-SPEC-014: UserStateTransition Functional Correctness

Commit	510c971	**Status**	Verified
Files	userStateTransition.circom		
Functions	UserStateTransition		

Description The circuit computes a user's state tree leaf with the input data and calculates the root of the merkle tree with the corresponding leaf and siblings. The root of the state tree and the input root of the epoch tree are then hashed together to produce the history tree leaf, which is then used to calculate the root of the history tree using the leaf and the input siblings.

Formal Definition The following shows the formal definition for the `UserStateTransition` template:

```
let user_state_transition =
  Circuit
    { name= "UserStateTransition"
    ; inputs=
        [ ("STATE_TREE_DEPTH", tnat)
        ; ("EPOCH_TREE_DEPTH", tnat)
        ; ("HISTORY_TREE_DEPTH", tnat)
        ; ("EPOCH_KEY_NONCE_PER_EPOCH", tnat)
        ; ("FIELD_COUNT", tnat)
        ; ("SUM_FIELD_COUNT", tnat)
        ; ("REPL_NONCE_BITS", tnat)
        ; ("from_epoch", tf)
        ; ("to_epoch", tf)
        ; ("identity_secret", tf)
        ; ("state_tree_indexes", tarr_t_k tf (v "STATE_TREE_DEPTH"))
        ; ("state_tree_elements", tarr_t_k tf (v "STATE_TREE_DEPTH"))
        ; ("history_tree_indices", tarr_t_k tf (v "HISTORY_TREE_DEPTH"))
        ; ("history_tree_elements", tarr_t_k tf (v "HISTORY_TREE_DEPTH"))
        ; ("attester_id", tf)
        ; ("data", tarr_t_k tf (v "FIELD_COUNT"))
        ; ( "new_data"
          , tarr_t_k
              (tarr_t_k tf (v "FIELD_COUNT"))
              (v "EPOCH_KEY_NONCE_PER_EPOCH") )
        ; ("epoch_tree_root", tf)
        ; ( "epoch_tree_elements"
          , tarr_t_k
              (tarr_t_k tf (v "EPOCH_TREE_DEPTH"))
              (v "EPOCH_KEY_NONCE_PER_EPOCH") )
        ; ( "epoch_tree_indices"
          , tarr_t_k
              (tarr_t_k tf (v "EPOCH_TREE_DEPTH"))
              (v "EPOCH_KEY_NONCE_PER_EPOCH") ) ]
    ; outputs=
        [ ( "history_tree_root"
          , t_r' (v "history_tree_indices")
```

```
              (v "history_tree_elements")
              identity_secret attester_id (v "from_epoch") (v "data") )
        ; ("state_tree_leaf", tf)
        ; ("epks", tarr_t_k tf (v "EPOCH_KEY_NONCE_PER_EPOCH")) ]
    ; dep= None
    ; body=
        elet "from_epoch_check"
          (call "Num2Bits" [zn 48; v "from_epoch"])
          (elet "to_epoch_check"
             (call "Num2Bits" [zn 48; v "to_epoch"])
             (elet "epoch_check"
                (call "GreaterThan" [zn 48; v "to_epoch"; v "from_epoch"])
                (elet "u0"
                   (assert_eq (v "epoch_check") f1)
                   (elet "attester_id_check"
                      (call "Num2Bits" [zn 160; v "attester_id"])
                      (elet "leaf_hasher"
                         (call "StateTreeLeaf"
                            [ v "FIELD_COUNT"
                            ; v "data"
                            ; identity_secret
                            ; attester_id
                            ; v "from_epoch" ] )
                         (elet "state_merkletree"
                            (call "MerkleTreeInclusionProof"
                               [ v "STATE_TREE_DEPTH"
                               ; v "leaf_hasher"
                               ; v "state_tree_indexes"
                               ; v "state_tree_elements" ] )
                            (elet "history_leaf_hasher"
                               (call "Poseidon"
                                  [ z2
                                  ; const_array tf
                                      [v "state_merkletree"; v "epoch_tree_root"]
                                  ] )
                               (elet "history_merkletree"
                                  (call "MerkleTreeInclusionProof"
                                     [ v "HISTORY_TREE_DEPTH"
                                     ; v "history_leaf_hasher"
                                     ; v "history_tree_indices"
                                     ; v "history_tree_elements" ] )
                                  (make
                                     [ v "history_merkletree"
                                     ; f0
                                     ; consts_n
                                         (v "EPOCH_KEY_NONCE_PER_EPOCH")
                                         f0 ] ) ) ) ) ) ) ) ) ) ) }
```

Formal Specification The following shows the formal specification for the `UserStateTransition` template:

```
let identity_secret = v "identity_secret"
let reveal_nonce = v "reveal_nonce"
let attester_id = v "attester_id"
let epoch = v "epoch"
let nonce = v "nonce"
let u_hasher z init = unint "MrklTreeInclPfHash" [z; init]
let u_zip xs ys = unint "zip" [xs; ys]
let u_state_tree_leaf a b = unint "u_state_tree_leaf" [a; b]
let data_drop_1 data = drop data z1

let t_r’ path_index path_elements identity_secret attester_id epoch data =
  tfq
    (qeq nu
       (u_hasher
          (u_zip path_index path_elements)
          (u_poseidon z2
             (const_array tf
                [ u_hasher
                    (u_zip (v "state_tree_indexes") (v "state_tree_elements"))
                    (u_poseidon z3
                       (const_array tf
                          [ identity_secret
                          ; fadd attester_id (fmul (fpow f2 (zn 160)) epoch)
                          ; u_state_tree_leaf (data_drop_1 data) (get data z0)
                          ] ) )
                ; v "epoch_tree_root" ] ) ) ) )
```