NEBRA UPA v1.1.0 Protocol Specification

Nebra Labs

2024

Overview

Application developers register VKs for their circuits with the UPA contract. Each VK is assigned a *circuit id* $\mathsf{circuitId}$ (the poseidon hash of the VK) and shared with off-chain aggregators (via events).

Application Clients submit proofs and PIs to the UPA contract as tuples $(\pi, \mathsf{PI}, \mathsf{circuitId})$, where `proof` is expected to be a proof that PI is an instance of the circuit with *circuit id* $\mathsf{circuitId}$.

A single call to the contract submits an *ordered list* $(\pi_i, \mathsf{PI}_i, \mathsf{circuitId}_i)_{i=0}^{n-1}$ (of any size n up to some implementation-defined maximum N) of these tuples. This ordered list of tuples is referred to as a *Submission*. Submissions of more than 1 proof allow the client to amortize the cost of submitting proofs. Note that there is no requirement for the $\mathsf{circuitId}_i$s to match, namely, a single *Submission* may contain proofs for multiple application circuits.

Each tuple in the submission is assigned:

- $\mathsf{proofId}$ - a unique *proof id* (equal to the Keccak hash of the circuit ID and PIs), and
- $\mathsf{proofIndex}$ - a *proof index* (a simple incrementing counter).

The submission is assigned:

- a *Submission Id* $\mathsf{submissionId}$, computed as the Merkle root of the list of $\mathsf{proofId}_i$s, padded to the nearest power of 2 with `bytes32(0)`.
- a *submission index* $\mathsf{submissionIndex}$, a simple incrementing counter of submissions, used later for censorship resistance.

Note that:

- for submissions that consist of a single proof, $\mathsf{proofId}_0 == \mathsf{submissionId}$, and the submitted proof can be referenced by $\mathsf{proofId}_0$, whereas
- for submissions of multiple proofs, each proof is referred to by $\mathsf{submissionId}$ and an index (or *location*) of the proof within the submission. Where required, a Merkle proof can be used to show that a proof with $\mathsf{proofId}_i$ is indeed at the given index within the submission $\mathsf{submissionId}$.

The proof and public input data is not stored on-chain, but is emitted as Events for *Aggregators* to monitor and receive. The contract stores information about the submission (including $\mathsf{submissionIndex}$, n and some further metadata), indexed by the submission Id.

Aggregators aggregate *batches* of proofs with *increasing* proof index values. In the case where invalid proofs have been submitted, aggregators may skip *only* invalid proofs. Aggregators that skip valid proofs will be punished (see below).

As aggregated batches of proofs are received and verified by the UPA contract, the corresponding proof ids are marked as verified. Note that, for proofs that are part of a multi-proof submission, the contract records the fact that the proof at location i of submission $\mathsf{submissionId}$ was verified.

An application client can then submit a transaction to the application circuit (optionally with some `ProofReference` metadata), and the application circuit can verify the existence of an associated ZKP as follows:

- Application contract computes the public inputs for the proof, exactly as it would in the absence of UPA
- Application contract passes the public inputs PI, the circuit Id $\mathsf{circuitId}$, and any metadata to the UPA contract.
- The UPA contract computes $\mathsf{proofId} = \mathsf{keccak}(\mathsf{circuitId}, \mathsf{PI})$ from the public inputs.
  - If the proof was submitted by itself, $\mathsf{proofId}$ is equal to the submission ID, and the contract can immediately check whether a valid proof has been seen as part of an aggregated proof.

- If the proof was part of a multi-proof submission, the metadata includes the $\mathsf{submissionId}$, index i of the proof within the submission $\mathsf{submissionId}$, and a Merkle proof that $\mathsf{proofId}$ is indeed the i-th leaf of the submission. After checking this Merkle proof, the contract can immediately verify that proof i of submission $\mathsf{submissionId}$ has been seen as part of an aggregated proof batch.

- The UPA contract returns 1 if it has a record of a valid proof for $(\mathsf{circuitId}, \mathsf{proofId})$, and 0 otherwise.

Protocol

Circuit registration

The application developer submits a transaction calling the `registerVK` method on the UPA contract, passing the verification key VK.

The circuitId $\mathsf{circuitId}$ for the circuit is computed as

$$\mathsf{circuitId} = \mathsf{compute\_circuit\_id}(\mathsf{VK}) = \text{poseidon}(\mathsf{DT}_{\mathsf{cid}}||\mathsf{VK})$$

where $\mathsf{DT}_{\mathsf{cid}}$ denotes a domain tag derived from a string describing the context, such as "Saturn v1.0.0 CircuitId" (See the Universal Batch Verifier specification for details.)

(We assume VK is serialized using SnarkJS or following the exactly the same protocol of SnarkJS).

VK is stored on the contract (for censorship resistance), indexed by $\mathsf{circuitId}$, and aggregators (who are assumed to be monitoring the contract) are notified via an event.

> NOTE: The poseidon hash is expensive to compute in the EVM, but this operation is only performed once at registration time. This $\mathsf{circuitId}$ will be used to reference the circuit for future operations.

Proof submission

The *App Client* creates the parameters for its smart contract as normal, including one or more proofs π_i and public inputs PI_i. It then passes these, along with the relevant (pre-registered) circuit Ids $\mathsf{circuitId}_i$, to the `submit` method on the UPA contract, paying the aggregation fee in ether:

```
contract Upa
{
    ...
    function submit(
            uint256[] calldata circuitId,
            Proof[] calldata proof,
            uint256[][] calldata publicInputs)
        external
        payable;
    ...
}
```

The `Upa.submit` method:

- computes $\mathsf{proofId}_i = \mathsf{keccak}(\mathsf{circuitId}_i, \mathsf{PI}_i)$ for $i = 0, \ldots, n-1$.
- computes a `proofDigest` $\mathsf{proofDigest}_i$ for each proof, as $\mathsf{keccak}(\pi_i)$
- computes the submission Id $\mathsf{submissionId}$ as the Merkle root of the list $(\mathsf{proofId}_i)_{i=0}^{n-1}$ (padded as required to the nearest power of 2)
- computes the `digestRoot` as the Merkle root of the list $(\mathsf{proofDigest}_i)_{i=0}^{n-1}$ (again padded as required to the nearest power of 2)
- rejects the tx if an entry for $\mathsf{submissionId}$ already exists
- assigns a $\mathsf{submissionIndex}$ to the submission (using a single incrementing counter)
- assigns a $\mathsf{proofIndex}_i$ to each (π_i, PI_i) (using a single incrementing counter)
- emits an event for each proof, including $(\mathsf{circuitId}_i, \pi_i, \mathsf{PI}_i, \mathsf{proofIndex}_i)$
- updates contract state to record the fact that a submission with id $\mathsf{submissionId}$ has been made, recording `digestRoot`, $\mathsf{submissionIndex}$, n and the block number at submission time.

Note: Proof data itself does not appear in the input data used to compute `proofId`. This is because, when the proof is verified by the application, the application does not have access to (and does not require) any proof data. Thereby, the application is in fact verifying the *existence* of some proof for the given circuit and public inputs.

Note: Application authors must ensure that the public inputs to their ZKPs contain some random or unpredictable elements (and in general this will already be the case for sound protocols, in order to prevent replay attacks). If the set of public inputs can be predicted by a malicious party, that malicious party can submit an invalid proof for the public inputs, preventing submission of further (valid) proofs for that same set of public inputs.

Aggregated proof submission

Aggregators submit aggregated proofs to the `Upa.verifyAggregatedProof` method, proving validity of a set of previously submitted application proofs. In return, they can claim batch submission fees.

```
function verifyAggregatedProof(
        bytes calldata proof,
        bytes32[] calldata proofIds,
        SubmissionProof[] calldata submissionProofs)
    external;
```

> `submissionProof` is an array of 0 or more proofs, each showing that some of the entries in `proofIds` belong to a specific multi-proof submission. These are required as we do not have a map from `proofId` to `submissionId` or `submissionIdx`. See the algorithm below for details.

The UPA contract:

- checks that `proof` is valid for `proofIds`
- for each proofId in `proofIds`,
  - check that proofId has been submitted to the contract, and that proofs appear in the aggregated batch in the order of submission (see below)
  - mark proofId as valid (see below)
  - emit an event indicating that proofId has been verified

Specifically, the algorithm for verifying submission (in the correct order) of `proofIds`, and marking them as verified, is as follows.

State: the contract holds

- a dynamic array `uint16[] numVerifiedInSubmission` of counters, where the i-th entry corresponds to the number of proofs that have been verified (in order) of the submission with $\mathsf{submissionIndex} == i$
- the submission index `nextSubmissionIdxToVerify` of the next submission from which proofs are expected.

Given a list of `proofIds` and `submissionProofs`, the contract verified that `proofIds` appear in submissions as follows:

- For each proofId in `proofIds`:
  - Attempt to lookup the submission data (see "Proof Submission") for a submission with Id proofId. If such a submission exists:
    - The proof was submitted as a single-proof submission. The contract extracts the submissionIndex from the submission data and ensures that submissionIndex is greater than or equal to `nextSubmissionIdxToVerify`. If not, reject the transaction.
    - The entry `numVerifiedInSubmission[` submissionIndex `]` should logically be 0 (this can be sanity checked by the contract). Set this entry to 1
    - update `nextSubmissionIdxToVerify` in contract state
  - Otherwise (if no submission data was found for submissionId = proofId)
    - the proof is expected to be part of a multi-proof submission with submissionIndex $\geq$ `nextSubmissionIdxToVerify`.
      - Note that if a previous aggregated proof verified some subset, but not all, of the entries in the submission, `nextSubmissionIdxToVerify` would still refer to the partially verified submission at this stage. In this case, `numVerifiedInSubmission[` submissionIndex `]` should contain the number of entries already verified.
    - Take the next entry in `submissionProofs`. This includes the following information:
      - the submissionId for the submission to be verified
      - a Merkle "interval" proof for a contiguous set of entries from that submission.
- Determine the number `m` of entries in `proofIds`, including the current proofId, that belong to this submission, as follows:
  - Let `numProofIdsRemaining` be the number of entries (including proofId) still unchecked in `proofIds`.
  - Look up the submission data for submissionId, in particular submissionIndex and n.
  - Let `numUnverifiedFromSubmission =`n`- numVerifiedInSubmission[` submissionIndex `]`.
  - The number `m` of entries from `proofIds` to consider as part of submissionId is given by `Min(numUnverifiedFromSubmission, numProofIdsRemaining)`.

- Use the submission Id submissionId and the Merkle "interval" proof from the submission proof, to check that the `m` next entries from `proofIds` (including proofId) indeed belong to the submission submissionId. Reject the transaction if this check fails.
- Increment the entry `numVerifiedInSubmission[ submissionIndex ]` by `m`, indicating that `m` more proofs from the submission have been verified.
- update `nextSubmissionIdxToVerify` in contract state, if all proofs from this submission have been verified

See the `UpaVerifier.sol` file for the code corresponding to the above algorithm

Proof verification by the application

The application client now creates the transaction calling the application's smart contract to perform the business logic. Since the proof has already been submitted to the UPA, the proof is not required in this transaction. If the proof was submitted as part of a multi-entry submission, the client must compute and send a `ProofReference` structure, indicating which submission the proof belongs to, and its "location" (or index) within it.

The application contract computes the public inputs, exactly as it otherwise would under normal operation, and queries the UPA contract (using the proofRef if given) to confirm the existence of a corresponding verified proof.

For proofs from single-entry submissions, the UPA provides the entry point:

```
function isVerified(
        uint256 circuitId,
        uint256[] calldata publicInputs)
    external
    view
    returns (bool);
```

For proofs from multi-entry submissions:

```
function isVerified(
        uint256 circuitId,
        uint256[] calldata publicInputs,
        ProofReference calldata proofRef)
    external
    view
    returns (bool);
```

The UPA contract:

- computes proofId from the public inputs
- (using the `ProofReference` if necessary) confirms that proofId belongs to a submission submissionId and reads the submission index submissionIndex.
- given submissionIndex and the index `i` of the proof within the submission (taken from the `ProofReference`, or implicitly `0` for the single-entry submission case), the existence of a verified proof is given by the boolean value: `numVerifiedInSubmission[submissionIndex] > i`

Censorship resistance

A censorship event is considered to have occured for a submission with Id submissionId (with submission index submissionIndex, consisting of n entries) if all of the following are satisfied:

- a submission with Id submissionId has been made, and **all** proofs in the submission are valid for the corresponding public inputs and circuit Ids
- some of the entries in submissionId remain unverified, namely
  - `numVerifiedInSubmission[submissionIndex] <`n
- one or more proofs from submission with index greater than submissionIndex (the submission index of the submission with id submissionId) have been included in an aggregated batch. Namely, there exists $j >$ submissionIndex s.t. `numVerifiedInSubmission[`j`] > 0` (or alternatively `nextSubmissionIdxToVerify` $>$ submissionIndex)

Note that, if one or more entries in a submission are invalid, aggregators are not obliged to verify any proofs from that submission.

Censorship by an *Aggregator* can be proven by a *claimant*, by calling the method:

```
function challenge(
    uint256 circuitId,
    Proof calldata proof,
    uint256[] calldata publicInputs,
    bytes32 submissionId,
    bytes32[] proofIdMerkleProof,
    bytes32[] proofDigestMerkleProof,
) external;
```

providing:

- the **valid** tuple $(\mathsf{circuitId}, \pi, \mathsf{PI})$, or `circuitId`, `proof` and `publicInputs`, the claimed next unverified entry in the submission
- submissionId or `submissionId`
- A Merkle proof that $\mathsf{proofId}_i$ (computed from $\mathsf{circuitId}_i$ and PI_i) belongs to the submission (at the "next index" - see below)
- A Merkle proof that π_i belongs to the submission's `proofDigest` entry (at the "next index" - see below)

Here "next index" is determined by the `numVerifiedInSubmission` entry for this submission. That is, proofs that have been skipped by the aggregators must be provided in the order that they occur in the submission.

On receipt of a transaction calling this method, the contract:

- checks that the conditions above hold and that the provided proof has indeed been skipped
- checks the claimant is the original submitter
- looks up the verification key VK using circuitId and performs the full proof verification for $(\mathsf{VK}, \pi, \mathsf{PI})$. The transaction is rejected if the proof is not valid.
- increments the stored count `numVerifiedInSubmission[submissionIndex]`

The aggregator is punished only when all proofs in the submission have been shown to be valid. As such, after the above, the contract:

- checks the condition `numVerifiedInSubmission[submissionIndex] == n` (where `n` is the number of proofs in the original submission submissionId).
- if this final condition holds then validity of all proofs in the submission has been shown and the aggregator is punished.

Note: `proofDigest` is used here to prevent malicious clients from submitting invalid proofs, forcing aggregators to skip their proofs, and then later provide valid proofs for the same public inputs. This would otherwise be an attack vector since `proofId` is not dependent on the proof data.

> TODO: the above assumes a single aggregator. For multiple aggregators, we must record extra information in order to determine which aggregator skipped a valid proof. We may need to introduce some time interval during which claims can be made (e.g. claims must be made before the proof index increases more than 2^12, say). Similarly, if penalties are to be paid from stake, aggregators should have an "unbonding period" of at least this interval.

Circuit Statements

Batches of n application proofs are verified in a *batch verify* circuit using batched Groth16 verification.

A *keccak circuit* computes all proofIds of application proofs appearing in the *batch verify* proof, along with a *final digest* (the keccak hash of these proofIds, used to reduce the public input size of the outer circuit below).

A collection of N *batch verify* proofs along with the *keccak* proof for their proofIds and *final digest* is verified in an *outer* circuit.

On-chain verification of an outer circuit proof thereby attests to the validity of $n \times N$ application proofs with given proofIds.

- n - inner batch size. Application proofs per *batch verify* circuit.
- N - outer batch size. Number of *batch verify* circuits per outer proof.
- L - the maximum number of public inputs for an application circuit.

Batch Verify Circuit: Groth16 batch verifier

The batch verify circuit corresponds to the following relation:

- *Public inputs*:
  - $(\ell_i, \mathsf{circuitId}_i, \overline{\mathsf{PI}}_i)_{i=1}^n$ where
    - $\mathsf{PI}_i = (x_{i,j})_{j=1}^{\ell_i}$ is the public inputs to the i-th proof
    - $\overline{\mathsf{PI}}_i = \mathsf{PI}_i|\{0\}_{j=\ell_i+1}^L$ is PI_i after zero-padded to extend it to length L
- *Witness values*:
  - $\overline{\mathsf{VK}}_i$ - application verification keys, each padded to length L
  - $(\pi_i)_{i=1}^n$ - application proofs
- *Equivalent Statement*:
  - $\mathsf{circuitId}_i = \mathsf{compute\_circuit\_id}(\text{truncate}(\ell_i, \overline{\mathsf{VK}}_i))$
  - $\overline{\mathsf{PI}}_i = \text{truncate}(\ell_i, \overline{\mathsf{PI}_i})|\{0\}_{j=\ell_i+1}^L$
  - $\texttt{Groth16.Verify}(\overline{\mathsf{VK}_i}, \pi_i, \overline{\mathsf{PI}_i}) = 1$ for $i = 1, \dots, n$ (batched G16)
  - where
    - $\text{truncate}(\ell, \overline{\mathsf{VK}})$ is the truncation of the size L verification key $\overline{\mathsf{VK}}$ to a verification key of size ℓ, and
    - $\text{truncate}(\ell, \overline{\mathsf{PI}})$ is the truncation of the public inputs to an array of size ℓ

Keccak Circuit: ProofIDs and Final Digest

Computes the $\mathsf{proofId}$ for each entry in each application proof in one or more verify circuit proofs.

- *Public inputs*:
  - $c^*, (\ell_i, \mathsf{circuitId}_i, \overline{\mathsf{PI}}_i)_{i=1}^{n\times N}$ where
    - $\mathsf{PI}_i = (x_{i,j})_{j=1}^{\ell_i}$ is the public inputs to the i-th proof
    - $\overline{\mathsf{PI}}_i = \mathsf{PI}_i|\{0\}_{j=\ell_i+1}^L$ is PI_i after zero-padded to extend it to length L
    - $c^* = (c_1^*, c_2^*)$ (32 byte *final digest*, represented by two field elements)
- *Witness values*: (none)
- *Statement*:
  - $c_i = \mathsf{keccak}(\mathsf{circuitId}_i || \text{truncate}(\ell_i, \overline{\mathsf{PI}}_i))$
  - $c^* = \mathsf{keccak}(c_1 || c_2 || \dots || c_{n\times N})$

Outer Circuit: Recursive verification of Batch Verifier and Keccak circuits

This circuit checks the validity of N *batch verify* proofs ${\pi_{\text{bv}}}^{(j)}, j = 1, \dots N$ as well as a single corresponding *keccak* proof π_{keccak}.

- *Public Inputs*:
  - c^* - 32-byte *final digest*, encoded as $(c_1, c_2) \in \mathbb{F}_r^2$
  - $(L, R) \in \mathbb{G}_1^2$ - overall KZG accumulator, encoded in $(\mathbb{F}_r)^{12}$ where 12 comes from $4\times$ `num_limbs`.
- *Witness values*:
  - $(\ell_{j,i}, \mathsf{circuitId}_{j,i}, \overline{\mathsf{PI}}_{j,i}, \mathsf{proofId}_{j,i})$ for $i = 1, \dots, n$, $j = 1, \dots, N$, the number of public inputs, the circuit ID, padded public inputs and proof ID for the i-th application proof in the j-th BV proof.
  - $(\pi_{\text{bv}}^{(j)})$ for $j = 1, \dots, N$ BV proofs
  - π_{keccak} the keccak proof for public inputs
    - c^*, and
    - $(\ell_{1,1}, \mathsf{circuitId}_{1,1}, \overline{\mathsf{PI}}_{1,1}), (\ell_{1,2}, \mathsf{circuitId}_{1,2}, \overline{\mathsf{PI}}_{1,2}), \dots, (\ell_{1,n}, \mathsf{circuitId}_{1,n}, \overline{\mathsf{PI}}_{1,n}),$
    - $(\ell_{2,1}, \mathsf{circuitId}_{2,1}, \overline{\mathsf{PI}}_{2,1}), (\ell_{2,2}, \mathsf{circuitId}_{2,2}, \overline{\mathsf{PI}}_{2,2}), \dots, (\ell_{2,n}, \mathsf{circuitId}_{2,n}, \overline{\mathsf{PI}}_{2,n}),$
    - $\cdots$
    - $(\ell_{N,1}, \mathsf{circuitId}_{N,1}, \overline{\mathsf{PI}}_{N,1}), (\ell_{N,2}, \mathsf{circuitId}_{2,N}, \overline{\mathsf{PI}}_{N,2}), \dots, (\ell_{N,n}, \mathsf{circuitId}_{N,n}, \overline{\mathsf{PI}}_{N,n}),$
- *"Equivalent Statement"*: (actual statement is shown as multiple sub-statements, given below)
  - For each $j = 1, \dots, N$, $\pi_{\text{bv}}^{(j)}$ is a valid proof of the *batch verify* circuit, for public inputs $(\ell_{j,i}, \mathsf{circuitId}_{j,i}, \overline{\mathsf{PI}}_{j,i})_{i=1}^n$, namely:

$$\mathsf{SNARK}_{\text{BV}}.\mathsf{Verify}\left(\pi_{\text{bv}}^{(j)}, (\ell_{j,i}, \mathsf{circuitId}_{j,i}, \overline{\mathsf{PI}}_{j,i})_{i=1}^n, \mathsf{VK}_{\text{BV}}\right) = 1$$

  - Keccak proof is valid, and therefore c^* is the *final digest* for all application PIs and vk hashes, namely:

$$\mathsf{SNARK}_{\mathsf{keccak}}.\mathsf{Verify}\left(\pi_{\mathsf{keccak}}, c^*, (\ell_{j,i}, \mathsf{circuitId}_{j,i}, \overline{\mathsf{PI}_{j,i}})_{\substack{i=1,\dots,n\\ j=1,\dots,N}}, \mathsf{VK}_{\mathsf{keccak}}\right) = 1$$

- Actual Statement:

– "Succinct" Plonk verification ($\mathsf{SuccinctVerify}$) namely "GWC Steps 1-11" using Shplonk, without final pairing:

$$(L_j, R_j) = \mathsf{SuccinctVerify}\left(\pi_{\mathrm{bv}}^{(j)}, (\ell_{j,i}, \mathsf{circuitId}_{j,i}, \overline{\mathsf{PI}_{j,i}})_{i=1}^{n}, \mathsf{VK}_{\mathrm{BV}}\right) \quad \text{for } j = 1, \ldots N$$

$$(L_{N+1}, R_{N+1}) = \mathsf{SuccinctVerify}\left(\pi_{\mathsf{keccak}}, c^*, (\ell_{j,i}, \mathsf{circuitId}_{j,i}, \overline{\mathsf{PI}_{j,i}})_{\substack{i=1,\ldots,n \\ j=1,\ldots,N}}, \mathsf{VK}_{\mathsf{keccak}}\right)$$

$$(L, R) = \sum_{j=1}^{N+1} r^j (L_j, R_j)$$

for random challenge scalar r.

- Verification: given $(\pi_{\mathrm{outer}}, L, R, c^*)$, the on-chain verifier performs the following:
  – $(L_{\mathrm{outer}}, R_{\mathrm{outer}}) := \mathsf{SuccinctVerify}(\pi_{\mathrm{outer}}, L, R, c^*, \mathsf{VK}_{\mathrm{outer}})$
  – for random challenge scalar r', check that $e(L + r' L_{\mathrm{outer}}, [\tau]_2) \stackrel{?}{=} e(R + r' R_{\mathrm{outer}}, [1]_2)$

Note:

- The same witness values $\overline{\mathsf{PI}}_{i,j}$ are used in the *outer* circuit to verify $\pi_{\mathrm{bv}}^{(j)}$ and π_{keccak}, implying that c^* is indeed the commitment to all application public inputs and circuit IDs.
- The outer circuit does not include the final pairing checks, therefore its statement is not that the BV/Keccak proofs are *valid*, but rather that they have been correctly accumulated into a single KZG accumulator (L, R). Checking that $e(L + r' L_{\mathrm{outer}}, [\tau]_2) \stackrel{?}{=} e(R + r' R_{\mathrm{outer}}, [1]_2)$, for random scalar r', therefore implies their validity.