NEBRA UPA v1.2.0 Protocol Specification

Nebra Labs

2024

Overview

Application developers register VKs for their circuits with the UPA contract. Eack VK is assigned a unique *circuit id* $\mathsf{circuitId}$ and shared with an off-chain *aggregator* via events.

Application Clients submit proofs and public inputs (PIs) to the UPA contract as tuples $(\mathsf{circuitId}, \pi, \mathsf{PI})$, where π is expected to be a proof of knowledge that PI is a valid instance for the circuit with *circuit id* $\mathsf{circuitId}$. These tuples may be submitted on-chain, to the UPA contract, or off-chain, directly to the aggregator.

A single call to the contract submits an *ordered list* $(\mathsf{circuitId}_i, \pi_i, \mathsf{PI}_i)_{i=0}^{n-1}$ (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*. 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 $(\mathsf{circuitId}, \pi, \mathsf{PI})$ in the submission is assigned:

- $\mathsf{proofId}$ - a unique *proof id*, computed as the Keccak hash of the circuit ID and PIs, and

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)`. Note that we hash the entries of this Merkle tree, so the leaves actually hold $\mathsf{keccak}(\mathsf{proofId}_i)$. In particular, the padding leaves hold $\mathsf{keccak}(\texttt{bytes32(0)})$.
- a *submission index* $\mathsf{submissionIndex}$ (only for on-chain submissions), an incrementing counter of on-chain submissions, used later for censorship resistance.

Note that:

- for submissions that consist of a single proof, $\mathsf{submissionId} == \mathsf{keccak}(\mathsf{proofId}_0)$, 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. The aggregator monitors for transactions submitting proofs to the contract and pulls this information from the transaction calldata. The contract stores information about the submission (including $\mathsf{submissionIndex}$, n and some further metadata), indexed by $\mathsf{submissionId}$.

The aggregator puts together *batches* of proofs with *increasing* submission index values. The proofs in a batch must be ordered exactly as they appear within submissions. Aggregated batches do not need to align with submissions- a batch may contain multiple submissions, and a submission may span multiple batches. Both on-chain and off-chain submissions can be aggregated in the same batch. If a submission contains any invalid proofs, it is considered *invalid*. The aggregator may skip *only* invalid submissions. If the aggregator skips a valid submission, then they will be punished (see below).

Once a submission is verified by the UPA contract, its submission id is marked as verified. Applications can confirm that an individual proof id is verified by providing a `ProofReference`, which is a Merkle proof that the proof id was included in a verified submission. Note that for proofs in a multi-proof submission with $\mathsf{submissionId}$, the contract does not mark the proof as verified until the entire submission has been verified.

Once the UPA contract marks a proof (or the submission containing a proof) as verified, an application client can submit a transaction to the application contract (optionally with some `ProofReference` metadata), and the application contract 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 submits the circuit id $\mathsf{circuitId}$, public inputs PI, and a `ProofReference` (where required) to the UPA contract.
- The UPA contract computes $\mathsf{proofId} = \mathsf{keccak}(\mathsf{circuitId}, \mathsf{PI})$ from the public inputs and then checks that `ProofReference` contains a valid Merkle proof that $\mathsf{proofId}$ belongs to a verified submission.
- The UPA contract returns 1 if it has a record of a valid proof for $(\mathsf{circuitId}, \mathsf{proofId})$, and 0 otherwise.

Application contracts can also verify the existence of multiple ZKPs belonging to the same submission. In this case:

- Application contract computes an array of public inputs $[\mathsf{PI}_i]$ where the i-th entry corresponds to the i-th proof of a submission $\mathsf{submissionId}$.
- Application contract submits an array of tuples $[(\mathsf{circuitId}_i, \mathsf{PI}_i)]$ to the UPA contract.
- The UPA contract computes the (unique) $\mathsf{submissionId}$ corresponding to the submitted circuit ids and public inputs.
- The UPA contract returns 1 if it has verified the submission $\mathsf{submissionId}$ (i.e. it has verified all of the proofs within $\mathsf{submissionId}$), and 0 otherwise.

Note that in this case, there is no need to submit a `ProofReference`.

Protocol

Circuit registration

Before submitting proofs on-chain, the application developer submits a transaction calling the `registerVK` method on the UPA contract, passing the verification key VK. Proof submitted off-chain do not need to register their VK with the contract.

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

$$\mathsf{circuitId} = \mathsf{keccak}(\mathsf{DT}_{\mathsf{cid}} || \mathsf{VK})$$

where $\mathsf{DT}_{\mathsf{cid}}$ denotes a domain tag derived from a string describing the context, such as `UPA Groth16 circuit id` (See the Universal Batch Verifier specification for details.)

VK is stored on the contract (for censorship resistance) in a mapping indexed by $\mathsf{circuitId}$, and the aggregator is notified via an event. This $\mathsf{circuitId}$ will be used to reference the circuit for future operations.

Proof submission

UPA supports two kinds of submission: on-chain and off-chain. On-chain submissions come with a strong censorship resistance guarantee- the aggregator can be penalized if it skips a valid submission. Off-chain submissions have no gas overhead for the proof submitter, but don't come with the same censorship resistance guarantees as those submitted on-chain.

Submitting proofs on-chain

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 circuitIds,
            Proof[] calldata proofs,
            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)
- computes the `proofDataDigest` as the keccak digest of `digestRoot` and `msg.sender`
- rejects the tx if an entry for $\mathsf{submissionId}$ already exists
- assigns a $\mathsf{submissionIndex}$ to the submission (using a single incrementing counter)
- emits an event for each submission containing the *Submission Id* $\mathsf{submissionId}$
- updates contract state to record the fact that a submission with id $\mathsf{submissionId}$ has been made, recording `proofDataDigest` (which is bound to `msg.sender`), $\mathsf{submissionIndex}$, the number of proofs in the submission 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 element that is hard to compute without the corresponding private witness (and in general this will already be the case for sound protocols, in order to prevent replay attacks). If the set of public inputs in a submission can be predicted by a malicious party, that malicious party can send an invalid submission for these public inputs, preventing on-chain submission of further (valid) proofs for that same set of public inputs.

Submitting proofs off-chain

Alternatively, proofs can be submitted out-of-band directly to the aggregator. These proofs are aggregated in an order independent of the on-chain queue, and therefore do not have an associated submission index $\mathsf{submissionIndex}$. The circuit id $\mathsf{circuitId}$, proof id $\mathsf{proofId}$ and a submission id $\mathsf{submissionId}$ are calculated in the same way as for proofs submitted on-chain.

Aggregated proof submission

There is a single (permissioned) *Aggregator* that submits aggregated proofs to the `Upa.verifyAggregatedProof` method. Each aggregated proof attests to the validity of a batch of application proofs. In return, the aggregator can claim submission fees (for on-chain submissions). An aggregated batch may contain proofs from both on-chain and off-chain submissions, as well as *dummy proofs* which are used to fill partial batches.

```
function verifyAggregatedProof(
        bytes calldata proof,
        bytes32[] calldata proofIds,
        uint16 numOnchainProofs,
        SubmissionProof[] calldata submissionProofs,
        uint256 offChainSubmissionMarkers)
    external;
```

`proof` - An aggregated proof for the validity of this batch.

`proofIds` is the list of proofIds that are verified by the aggregated proof `proof`. These are assumed to be arranged in the order: [On-chain, Dummy, Off-chain]. Furthermore, it is assumed that if there are dummy proofIds in this batch, these appear after the last proof in a submission. I.e. where dummy proof ids are used, the on-chain proof ids do not end with a partial submission.

`numOnChainProofs` - The number of proofIds that were from on-chain submissions. This count includes dummy proofs.

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

`offChainSubmissionMarkers` represents a `bool[]` marking each off-chain member of `proofIds` with a 0 or 1. A proofId is marked with a 1 precisely when the proofId is the last one in an off-chain submission. This `bool[]` is packed into a `uint256` to compress calldata.

The UPA contract:

- checks that `proof` is valid for `proofIds`

- for each on-chain proofId in `proofIds`,
  - skip proofId if it corresponds to a dummy proof,
  - 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)
  - if proofId is the last proof in a submission submissionId, emit an event indicating that the submission submissionId has been verified
- for each off-chain proofId in `proofIds`,
  - adds proofId to the storage array `bytes32[] currentSubmissionProofIds`.
  - if `offChainSubmissionMarkers` indicates that proofId is the last proofId in a submission, then the contract calculates the submissionId submissionId for the proofs in `currentSubmissionProofIds`. Then the contract resets the array `currentSubmissionProofIds` and updates the map entry `numVerifiedAtBlock[` submissionId `]` to hold the current block number.

Marking on-chain submissions as verified

In more detail, the algorithm for verifying on-chain submissions (in the correct order) of `proofIds`, and marking them as verified, is as follows.

On-chain Submission 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 submissionIndex $== i$
- the submission index `nextSubmissionIdxToVerify` of the next submission from which proofs are expected.

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

- For each proofId in `proofIds`:
  - If proofId corresponds to a dummy proof, skip ahead to the proofs submitted off-chain, which start at index `numOnchainProofs`.
  - Attempt to lookup the submission data (see "Proof Submission") for a submission with Id keccak(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 = keccak(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 hashes of the `m` next entries from `proofIds` (including keccak(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. The contract can then either use a proofId computed from the public inputs, or use the public inputs themselves, to query the UPA contract (using the `ProofReference` if given) to confirm the existence of a corresponding verified proof.

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

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

function isProofVerified(bytes32 proofId) external view returns (bool);
```

For proofs from multi-entry submissions, the UPA provides entry points:

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

function isProofVerified(
        bytes32 proofId,
        ProofReference calldata proofReference
    ) external view returns (bool);
```

The UPA contract:

- receives proofId or computes proofId from the public inputs
- (using the `ProofReference` if necessary) confirms that proofId belongs to a submission submissionId.
- Checks if there was an on-chain submission for submissionId, and if so reads the stored submission index submissionIndex and the total number of proofs `numProofs` contained in the submission submissionId. If it finds that `numVerifiedInSubmission[submissionIndex] == numProofs` then the submission submissionId was verified, and therefore so was the proof proofId.
- Otherwise, if there is no verified on-chain submission for submissionId, the UPA contract checks for a verified off-chain submission for submissionId. Such a submssion exists if and only if `verifiedAtBlock[submissionId] > 0`.

The application contract can also look up the verification status of entire submissions by computing the corresponding (nested) array of public inputs. The contract can then either use a submissionId computed from this array, or the array itself, to query the submission's status in the UPA contract.

The UPA provides the entry points:

```
// If all proofs have the same circuitId.
function isSubmissionVerified(
    uint256 circuitId,
    uint256[][] memory publicInputsArray
) external view returns (bool);

function isSubmissionVerified(
    uint256[] calldata circuitIds,
```

```
    uint256[][] memory publicInputsArray
) external view returns (bool);

function isSubmissionVerified(
    bytes32 submissionId
) external view returns (bool);
```

The UPA contract:

- receives submissionId or computes submissionId from the public inputs
- Looks up the number of proofs `numProofsInSubmission` in submissionId and then checks if `numVerifiedInSubmission[submissionIndex] = numProofsInSubmission`.
- If no verified on-chain submission is found for submissionId, check if there is a verified off-chain submission for submissionId. Such a submssion exists if and only if `verifiedAtBlock[submissionIndex] > 0`.

Censorship resistance for on-chain submissions

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, the aggregator is not obliged to verify any proofs from that submission.

Censorship by the *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 aggregator 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 the aggregator to skip their proofs, and then later providing valid proofs for the same public inputs. This would otherwise be an attack vector since `proofId` is not dependent on the proof data.

Note: The claimed value of `proofDigest` is computed from the Merkle proof sent by the challenger. This is checked by computing `keccak(claimedProofDigest, msg.sender)` and checking that the resulting value matches `proofDataDigest` in the submission.

Collecting Aggregation Fees

On-chain aggregation fees

The application contract pays an aggregation fee at submission time. These fees are held in the UPA contract. In order for the aggregator to claim the fees for a given submission, the UPA contract must have verified that submission.

The aggregator collects fees in two steps. First it calls

```
function allocateAggregatorFee(uint64 lastSubmittedSubmissionIdx)
```

which stores the current value of `lastSubmittedSubmissionIdx` and allocates all fees collected up to now to be claimable by the aggregator once it has verified the submission at `lastSubmittedSubmissionIdx` (which implies that all previous submissions have also been verified). Once the aggregator has done this, it can call

```
function claimAggregatorFee(
    address aggregator,
    uint64 lastVerifiedSubmissionIdx
)
```

to withdraw the previously allocated fees.

Off-chain aggregation fees

Compensation for aggregating off-chain submissions is out of scope for this protocol, and the aggregator is free to implement any strategy it wishes. We outline the following candidate implementation: The off-chain aggregator maintains contracts, to which submitters deposit funds to be used to pay for submission fees. Using signatures from submitter and aggregator, the aggregator may claim fees for verified submissions, and submitter may claim reimbursement / penalties for submissions not verified by some agreed deadline.

Circuit Statements

ZK circuits are used to verify the computations performed by the aggregator. They enforce the following soundness condition: For each proof ID $\mathsf{proofId}_i$ belonging to an aggregated proof, the aggregator has knowledge of a corresponding valid application proof. This condition is enforced by a system of recursive circuits, whose precise statements may be found in the UPA circuit specifications.

We note here the implication of this soundness property: If the aggregator has knowledge of a valid application proof for the proof ID $\mathsf{proofId}_i$, then by the soundness property of the zero-knowledge proof system used by the application, the user who generated that application proof must have knowledge of a valid witness for the circuit relation defined by the corresponding public inputs. Applications may therefore accept these inputs as valid, even without seeing the application proof.

(The above statements are understood to hold except with negligible probability in the security parameters.)