Webb Protocol: A cross-chain private application and governance protocol.

Drew Stone

drew@webb.tools

Abstract

In this paper, we present the Webb Protocol, a system for building and governing cross-chain applications that can support shared anonymity set functionality across a set of identical bridged systems on compatible blockchains. The Webb Protocol is composed of two major protocols that deal with storing, updating, and validating of data and state changes that occur on a bridge and that are relevant to replicate on each connected chain. State is efficiently verifiable through the use of merkle trees and privacy is provided using zero-knowledge proofs of membership. Together, one can create applications leveraging distributed state with private property testing capabilities. Both financial and non-financial applications are described as motivating examples within the paper.

Contents

1 Introduction

Private applications today are not as scalable as they can be. The main reason they are not scalable is because they create privacy independent of one another. A bridged privacy system, which gains privacy linearly or superlinearly to the number of connected systems, would create far more privacy than any solo-chain application could. Towards the goal of creating a bridged privacy system, we describe one that utilises and maintains a private dataset through a set of connected on-chain merkle trees. Each on-chain merkle tree utilizes a graph-like edge list for storing linked metadata to facilitate an interoperable and potentially private cross-chain application. We refer to the independent instances as anchors. Each anchor lives on a separate blockchain and we require that each blockchain on a bridge possess the cryptographic primitives necessary for enabling such a protocol.

We design two protocols that we call the Anchor System and the DKG Validation Protocol and provide symbolic specifications and practical implementations of these protocols. The Anchor System is responsible for storing state and updating a bridged set of anchors. The DKG Validation Protocol is responsible for validating the state changes using a distributed key generation protocol that generates threshold-signatures. We discuss trustless improvements to the validation protocol at the end.

2 Background

A graph $G = (V, E)$ is composed of a set of vertices V and edges $E = V \times V$. We let $\mathsf{neighbors} : V \longrightarrow V^*$ denote the neighbors of each $v \in V$.

We let $r \overset{\$}{\leftarrow} \mathbf{F}_p$ denote a random sampling of an element from a prime field with prime p. We let $H \in \mathcal{H}$ be a family of collision-resistant hash functions H.

We let $(\mathsf{pk},\mathsf{sk})$ denote a public and private key pair. We let this be arbitrary for generality. One can imagine this to be an ECDSA keypair or a El Gamal encryption keypair.

We will utilise arithmetic circuits to design the zero-knowledge membership circuit. We refer to the circuit as C. A zero-knowledge circuit expresses a program that we aim to generate zero-knowledge proofs for. Creating the proof involves an interactive protocol between a prover $\mathcal{P}$ and a verifier $\mathcal{V}$, wherein $\mathcal{P}$ aims to convince $\mathcal{V}$ of knowledge of a pair $(x, w) \in \mathcal{R}$. x is the statement, w is a private witness, and $\mathcal{R}$ is the relation of satisfiable statement and witness pairs. As we will discuss more in-depth later, each anchor will maintain a copy of a verifier $\mathcal{V}$ for the circuit expressing the specific relevant relation.

Definition 1 *A zkSNARK protocol for an arithmetic circuit C is composed of a trio of algorithms: (*$\textsc{Setup}$, $\textsc{Prove}$, $\textsc{Verify}$*).*

- *(pp)* $\longleftarrow \textsc{Setup}(1^\lambda, C)$ *generates a set of public parameters $\mathsf{pp} \in \mathbb{F}_p^k$ where k is a parameter derived from the circuit C.*
- $\pi \longleftarrow \textsc{Prove}(\mathsf{pp}, x, w)$ *generates a proof π using a statement x and witness w.*
- $b \longleftarrow \textsc{Verify}(\mathsf{pp}, x, \pi)$ *outputs a boolean value $b \in \{0, 1\}$ depending on if the proof π is satisfying for the statement x.*

3 Anchor System

The Anchor System is a bridging system for merkle trees and metadata. Each anchor contains an on-chain merkle tree that users insert data into, ideally as collision-resistant commitments to data. Each anchor contains an edge list possessing the latest header of neighboring anchors. The anchor implements an API for modifying each of the listed items – from the merkle tree to the edge list.

3.1 Purpose

The Anchor System serves a few primary purposes:

1. Maintain an up-to-date list of neighboring anchor metadata and expose an API to modify this state.
2. Maintain an on-chain verifier that verifies zero-knowledge proofs for an application-specific circuit relation.

3.2 Governance

The Anchor System has governance too. State changes touching neighboring anchor state are considered governance actions. If we want to modify data in the list of neighbors, then we must get the governance system to approve this state change.

We consider three main regimes of governance systems: proof based, signature based, or token weighted based governance system. They all represent ways of validating a governance action.

1. Proof based governance actions are able to execute with certain proofs, such as a zero-knowledge proof or a fraud proof or a light-client proof.
2. Signature based governance actions are able to execute with valid signatures, such as ECDSA signatures or threshold signatures.
3. Token weight governance actions are able to execute depending on a function of tokens voting in favor of the action.

3.3 State and API

We describe the required storage and API functionality for a minimal Anchor System implementation. The pseudocode in 2 will follow the syntax of Rust. The pseudocode illustrates the implementation of a single anchor that would live on one blockchain.

An anchor implementation must fulfill a few simple storage requirements. Namely, it must maintain the storage required for a merkle tree. It must then store at least the required data needed to rebuild and update the merkle root after new insertions. Beyond this, it is a developer's decision to augment the storage required for end application purposes.

An anchor implementation must also store a graph-like interface that allows the anchor to maintain a *view* of the bridge's state. An anchor only knows about the latest state of its neighbors that it has been told about. Updates to an anchor's local edge list is validated through a governance action such as one defined above. The data in an edge contains relevant data about the neighboring anchor such as the neighboring anchor's merkle root, the latest leaf insertion index, and other metadata that we remark on below. We encapsulate this data into an object we often refer to as an Edge or a EdgeMetadata.

Definition 2 *An* Edge *or* EdgeMetadata *is an abstract object composed of:*

1. *A* chain_id *representing a unique chain identifier for the blockchain where the neighboring anchor exists.*

2. *A* merkle_root *representing the neighboring anchor's merkle tree root, expressed as a byte array.*

3. *A* target_resource *representing a unique resource identifier for the neighboring anchor on its respective blockchain, expressed as a byte array.*

4. *A* nonce *representing the index of the last leaf insertion that mutated the tree into* merkle_root*.*

Definition 3 *An* EdgeList *is a bounded list of edges of type* Edge*.*

Definition 4 *An **anchor** must support at the minimum an API that enables merkle tree insertions, edge list updates, and queries as follows:*

- `insert`(leaf: Bytes) - *Allows inserting a leaf of type* Bytes *(a fixed-byte array) into the underlying merkle tree.*

- `update_edge`(edge: Edge) - *Allows updating an edge in the underlying* EdgeList *for the entry indexed at* edge.chain_id*.*

- `get_own()` $\longrightarrow$ Edge - *Retrieves and formats the anchor's edge data for the purposes of relaying to neighboring anchors* `update_edge` *function.*

- `get_neighbors()`$\longrightarrow$ EdgeList - *Retrieves the list of neighboring edges stored on the anchor. The edge list returned contains only the neighboring edges.*

3.4 Privacy

The Anchor System enables collaborative and shared privacy through the use of zero-knowledge proofs of membership of data in one-of-many merkle trees. The *many* merkle trees are taken to be the anchors' dedicated merkle tree root as well as its neighbors' roots, according to its view of the bridge's state. Using various structures of data for leaves in these merkle trees, a user can prove – from anywhere and privately – properties of their data in a privacy-preserving manner.

Privacy is achieved collaboratively because the zero-knowledge proof creates an interoperable anonymity set between the set of connected anchors. More explicitly, since users can prove properties of data without disclosing also *where* the data was inserted, it is possible that the data could have been inserted into *any* anchor. The maximum degree of privacy of a bridged set of anchors grows linearly with each new anchor on the bridge. All that is left is to incentivize privacy-preserving behaviour.

3.5 Anchor System Gadget

The zero-knowledge bridge gadget for each Webb Anchor System instance expresses the following functionality. While we remark on the generality of such a mechanism, we will use a specific instantiation of values to provide an explicit example to motivate the mechanism's utility:

```
type ChainId = u64;
type ResourceId = [u8; 32];
enum TargetSystem {
    ContractAddress: [u8; 20],
    StorageIdentifier: u32,
    ...
}

struct MerkleTree {
    depth: u8,
    root: [u8; 32],
    leaves: Vec<[u8; 32]>,
    layers: Vec<Vec<[u8; 32]>>,
};

struct EdgeMetadata = {
    src_chain_id: ChainId,
    merkle_root: [u8; 32],
    nonce: u32,
    target_resource: ResourceId,
};

struct Anchor {
    resource: ResourceId
    edges: HashMap<ChainId, EdgeMetadata>,
    merkle_tree: MerkleTree,
};

trait AnchorInterface {
    fn insert(&mut self, leaf: [u8; 32]);
    fn update_edge(&mut self, data: EdgeMetadata);
    fn get_own(&self) -> EdgeMetadata;
    fn get_neighbors(&self) -> Vec<EdgeMetadata>;
}

impl AnchorInterface for Anchor { ... }
```

Figure 1: Pseudocode interfaces for an Anchor implementation

3.5.1 Constraints/proofs for the Anchor System.

1. A proof of knowledge of the preimage of a commitment cm. For example,

$$\begin{aligned}\mathsf{preimage}_{\mathsf{cm}} &= \{i, v, \mathsf{pk}, r\} \\ \mathsf{cm} &= H(i, v, \mathsf{pk}, r) \equiv H(\mathsf{preimage}_{\mathsf{cm}})\end{aligned}$$

   $i \in \mathbb{Z}$ is an integer identifier for the commitment's destination chain, $v \in \mathbb{Z}$ is the value of the commitment, and $r \overset{\$}{\leftarrow} \mathbb{F}_p$.

2. A proof of knowledge of the preimage of a serial number $\mathsf{preimage}_{\mathsf{sn}}$ or nullifier $\mathsf{preimage}_{\mathsf{nullifier}}$, which we may use interchangeably. This is used to identify when a commitment has be used.

$$\begin{aligned}\mathsf{preimage}_{\mathsf{sn}} &= \{\mathsf{cm}, j, \sigma\} \\ \mathsf{sn} &= H(\mathsf{cm}, j, \sigma)\end{aligned}$$

   j indicates the index of cm in a merkle tree and $\sigma = H(\mathsf{sk}, j, \mathsf{cm})$ represents a hash-based signature.

3. A proof correct computation of a merkle root:

$$\mathsf{MR}_{\mathsf{cm}} := \mathsf{reconstruct}(\mathsf{cm}, \mathsf{path}(\mathsf{cm}), H)$$

   $\mathsf{path}(\mathsf{cm})$ specifies a merkle proof path for cm. Here $\mathsf{reconstruct}$ is a standard merkle tree reconstruction algorithm.

4. A proof of membership of $\mathsf{MR}_{\mathsf{cm}} \in \{\mathsf{MR}_1, \mathsf{MR}_2, \ldots, \mathsf{MR}_k\} = \mathbf{MR}$ for a fixed public set of merkle roots.

Together in zero-knowledge, these proofs/constraints allow us to verify that commitments are in one-of-many merkle trees without disclosing which merkle tree the commitment exists within. This has implications for a variety of new applications that can be built both privately and interoperably. Next, we define a basic gadget relation that describes these constraints together.

3.6 Anchor System Gadget Relations

Example 1 *A* ***Basic Anchor System Gadget*** *expresses a zkSNARK protocol for the relation $\mathcal{R}$ such that for public inputs and private witness $w = \{v, r, \mathsf{sk}, j, \mathsf{path}(\mathsf{cm})\}$:*

$$\mathcal{R} = \left\{ (x, w) \;\middle|\; \begin{array}{l} \mathsf{cm} := H(\mathsf{preimage}_{\mathsf{cm}}) \\ \sigma := H(\mathsf{preimage}_{\sigma}) \\ \mathsf{MR}_{\mathsf{cm}} := \mathsf{reconstruct}(\mathsf{cm}, \mathsf{path}(\mathsf{cm}), H) \\ x := \mathsf{MR}_{\mathsf{cm}} \in \mathbf{MR} \wedge \mathsf{sn} = H(\mathsf{preimage}_{\mathsf{sn}}) \end{array} \right\}$$

This basic gadget example leaves application-specific constraints as a separate matter. The purpose of this relation is to highlight the constraints that describe how we achieve interoperability and zero-knowledge.

As described, the only public inputs are the set of merkle roots, a unique chain identifier, a and serial number intended to be exposed. The set of merkle roots represents anchor instances on individual blockchains. Generating a proof indicates knowledge of an element in one of the blockchains within a set and opens up the possibility to effectively transfer messages across chain with zero-knowledge proofs. These proofs can describe various constraints allowing us to property test our private data.

3.6.1 Example: Variable Asset Anchor System

As a motivating and concrete example of an application, we will describe the variable asset Anchor System, an interoperable shielded pool protocol. We have made a variety of values of the commitment and serial numbers relevant for a variable and private asset transfer system. By variable, we mean that users can transfer arbitrary amounts of assets. By private asset transfer system, we mean a system that allows users to transfer funds privately between blockchains. This is akin to a shielded pool but interoperable, where users can transfer a variable amount of a single asset between many blockchains using zero-knowledge proofs.

For this application we will use the example values provided above and replicated below. There will also be a integer public amount variable, indicating an additional liquidity deposit or withdrawal into or from the shielded pool. Recall,

$$\begin{aligned} \mathsf{cm} &= H(i, v, \mathsf{pk}, r) \\ \mathsf{sn} &= H(\mathsf{cm}, j, \sigma) \\ \sigma &= H(\mathsf{sk}, j, \mathsf{cm}) \end{aligned}$$

The system mimics that of a shielded UTXO system albeit with cross-chain capabilities. We describe below using the notion of multiple inputs and outputs, denoted by the integers $\mathsf{INS} \in \mathbb{Z}$ and $\mathsf{OUTS} \in \mathbb{Z}$. When we index variables by these values, we mean variables from input and output collections respectively. We now describe additional constraints for this example application below.

3.6.2 Constraints/proofs for the variable anchor asset protocol.

1. A proof of knowledge of the preimage and correct computation of each output commitment $\mathsf{cm}_o,\ o \leq \mathsf{OUTS}$. For example,

$$\{\mathsf{cm}_o = H(i_o, v_o, \mathsf{pk}_o, r_o)\}_{o \leq \mathsf{OUTS}}$$

2. A proof of knowledge that the sum of a public amount and input UTXO amounts equals the sum of output UTXO amounts:

$$\mathsf{public\_amount} + \sum_{k \leq \mathsf{INS}} v_k = \sum_{o \leq \mathsf{OUTS}} v_o$$

3. A proof of knowledge of no duplicate serial numbers, i.e. that no input is duplicated more than once.

We can contextualise the general Anchor System constraints with these new constraints in a relation for this specific application. Note, when we iterate over INS and OUTS the values within are specifically related to inputs and outputs respectively.

Example 2 *A **Variable Asset Anchor System Gadget** expresses a zkSNARK protocol for the relation $\mathcal{R}$ such that for:*

- *Public inputs:*

$$\{\mathsf{public\_amount}, \mathbf{MR}, i, \{\mathsf{sn}_k\}_{k \le \mathsf{INS}}, \{\mathsf{cm}_o\}_{o \le \mathsf{OUTS}}\}$$

- *Private inputs/witness:*

$$\mathbf{w} = \{v_k, r_k, \mathsf{sk}_k, j_k, \mathsf{path}(\mathsf{cm}_k)\}_{k \le \mathsf{INS}} \bigcup \{v_o, r_o, i_o, \mathsf{sk}_o, j_o\}_{o \le \mathsf{OUTS}} :$$

- *Relation*

$$\mathcal{R} = \left\{ (x, \mathbf{w}) \;\middle|\; \begin{array}{l} \{\mathsf{pk}_k := H(\mathsf{sk}_k)\}_{k \le \mathsf{INS}} \\ \{\mathsf{cm}_k := H(i, v_k, \mathsf{pk}_k, r_k)\}_{k \le \mathsf{INS}} \\ \{\sigma_k := H(\mathsf{sk}_k, j_k, \mathsf{cm}_k)]\}_{k \le INS} \\ \{\mathsf{MR}_{\mathsf{cm}_k} := \mathsf{reconstruct}(\mathsf{cm}_k, \mathsf{path}(\mathsf{cm}_k)), H)\}_{k \le \mathsf{INS}} \\ \{x_k := \mathsf{MR}_{\mathsf{cm}_k} \in \mathbf{MR}\}_{k \le INS} \end{array} \right\}$$

We let x be defined as:

$$\begin{aligned} x := & \Big(\bigwedge_{k=1}^{\mathsf{INS}} x_k \Big) \\ & \wedge \Big(\bigwedge_{k=1}^{\mathsf{INS}} \mathsf{sn}_k = H(\mathsf{cm}_k, j_k, \sigma_k) \Big) \\ & \wedge \Big(\bigwedge_{o=1}^{\mathsf{OUTS}} \mathsf{cm}_o = H(i_o, v_o, \mathsf{pk}_o, r_o) \Big) \\ & \wedge \Big(\mathsf{public\_amount} + \sum_{k \le \mathsf{INS}} v_k = \sum_{o \le \mathsf{OUTS}} v_o \Big) \\ & \wedge \Big(\{\mathsf{sn}_k\}_{k \le \mathsf{INS}} \text{ are distinct} \Big) \end{aligned}$$

3.6.3 Example: Semaphore Anchor System

Semaphore [2] is a popular zero-knowledge protocol originally developed and maintained by the Ethereum Foundation, a non-profit body active in the Ethereum [3] project. It allows members of an on-chain community to create arbitrary anonymous signals using zero-knowledge proofs of membership in the community's identity set, represented as a merkle tree. Signals are general and allow the mechanism to extend to privacy-preserving polling, voting, and whistle-blowing applications.

Here, we consider the interoperable extension to this application. Instead of maintaining a single community identity set, we define our community across a set of identity sets, located on potentially many other blockchains. Each identity set has a potentially different mechanism for approving modifications to its internal state, yet as a whole the community agrees on these differences. For example, we may want to construct a cross-chain identity set allowing any member of a set of NFT communities to register. We may also want to maintain a cross-chain ERC20 token gated communities.

We maintain each identity set in a merkle tree and similarly maintain a set of neighboring anchors identity sets' merkle tree roots. We identify the set of merkle roots of all the communities as $\mathbf{MR}_{\mathcal{S}}$. Then, using the same rough gadget as defined above, a user can prove their membership of an identity on one-of-many anchors without disclosing what the identity is nor where it lives, just that it exists in the cross-chain identity set. A more thorough specification can be found in the appendix.

We define some terminology and their constructions below. We indicate identity parameters using the subscript of *id*. To generate an identity, we follow the convention chosen by Semaphore. A user must first sample a random identity trapdoor and identity nullifier. A user generates their identity secret as the hash of those two values. A user finally generates their identity commitment as the hash of the identity secret.

$$\begin{aligned}
\mathsf{trapdoor}_{id} &\xleftarrow{\$} \mathbf{F}_p \\
\mathsf{nullifier}_{id} &\xleftarrow{\$} \mathbf{F}_p \\
\mathsf{secret}_{id} &= H(\mathsf{trapdoor}_{id}, \mathsf{nullifier}_{id}) \\
\mathsf{cm}_{id} &= H(\mathsf{secret}_{id})
\end{aligned}$$

Example 3 *A **Semaphore Anchor System Gadget** expresses a zkSNARK protocol for the relation $\mathcal{R}$ such that for:*

- *Public inputs:*

$$\{\mathsf{external\_nullifier}, \mathsf{signal\_hash}, \mathbf{MR}\}$$

- *Private inputs/witness:*

$$\mathbf{w} = \{\mathsf{trapdoor}_{id}, \mathsf{nullifier}_{id}, \mathsf{path}(\mathsf{cm}_{id})\} :$$

- *Relation*

$$\mathcal{R} = \left\{ (x, \mathbf{w}) \;\middle|\; \begin{array}{l} \mathsf{cm}_{id} := H(H(\mathsf{trapdoor}_{id}, \mathsf{nullifier}_{id})) \\ \mathsf{MR}_{\mathsf{cm}_{id}} := \mathsf{reconstruct}(\mathsf{cm}_{id}, \mathsf{path}(\mathsf{cm}_{id}), H) \\ x := \mathsf{MR}_{\mathsf{cm}_{id}} \in \mathbf{MR} \end{array} \right\}$$

3.6.4 Example: Identity-based Variable Asset Anchor System

Composing zero-knowledge applications enables us to build for even more use cases. Using the two example applications above – the Semaphore Identity protocol and the Variable Asset protocol – we can design a cross-chain shielded pool application over a restricted identity set. This yields a private transaction system where only users with proofs of membership in a cross-chain identity system can transact.

3.7 The Validation Protocol

The Validation Protocol is the backbone of security for the Anchor System. The main goal here is to validate messages passed between Anchor System instances to achieve a given set of security goals. The method of validation can be arbitrary, in that we only require there to be a definitive boolean outcome of the validity of a message. Within a Webb Protocol instance this can be customized and updated throughout the lifecycle of the protocol's operation.

Messages that pass this filter are processed directly and trigger state changes. Therefore, it is paramount to understand a given implementation's target security and to utilise the validation mechanism to achieve that goal.

3.7.1 State and API

The Validation Protocol must provide a simple API for validating a message's validity, depending on the underlying validation mechanism. The state involved in each unique mechainsm may differ from one to another, so we leave discussion of the state involved to the examples. Consequently, in the most general case, there is no unique state relevant for this protocol.

Definition 5 *The **Validation Protocol** must expose an API that allows a message to be verified with a proof. The types of these input parameters are general and must be defined by the underlying mechanism chosen.*

- `validate`(message: Message, proof: Proof)$\longrightarrow \{0, 1\}$ - *Processes and validates a message and a proof and outputs a bit value indicating success or failure.*

3.7.2 A single-signer ECDSA validation protocol.

In the single-signer ECDSA validation protocol, a single ECDSA public key dictates the mechanism. That is, any message signed by this key will output 1 and 0 otherwise. This account must be replicated everywhere messages are relayed and processed.

- `validate`(message: Message, proof: EcdsaSignature) $\longrightarrow \{0, 1\}$

  1. The message is hashed using the `keccak_256` hash function, yielding `keccak_256`(message).
  2. Using an ECDSA elliptic-curve public key recovery algorithm, denoted `ecrecover`, we recover the uncompressed ECDSA public key.

$$\text{public\_key} = \texttt{ecrecover}(\text{proof}, \texttt{keccak\_256}(\text{message}))$$

  3. Lastly, we verify the equality against the single-signer public key.

$$\texttt{if}\ \text{public\_key} = \text{single-signer}\ \{\texttt{return 1}\}\ \texttt{else}\ \{\texttt{return 0}\}$$

3.7.3 A multi-signer ECDSA validation protocol.

In the multi-signer ECDSA validation protocol, multiple ECDSA public keys dictate the mechanism. That is, any message signed by at least a threshold t of keys will output 1 and 0 otherwise. These accounts must be replicated everywhere messages are relayed and processed. Similarly, the threshold must also be stored everywhere messages are processed.

- `validate`(message: Message, proof: Vec<EcdsaSignature>) $\longrightarrow \{0, 1\}$

1. The message is hashed using the `keccak_256` hash function, yielding `keccak_256`(message).
2. Using an ECDSA elliptic-curve public key recovery algorithm, denoted `ecrecover`, we recover the unique uncompressed ECDSA public keys from all signatures.

$$\mathbf{PK} = \mathsf{unique}(\{\mathtt{ecrecover}(\mathsf{proof}[i], \mathtt{keccak\_256}(\mathsf{message}))\}_{i \leq |\mathsf{proof}|})$$

3. We sum the number of unique valid keys against the equality relation using the indicator random variables and check against the threshold t.

$$\mathtt{if}\ \Big(\sum_{\mathsf{pk} \in \mathbf{PK}} \mathbb{1}[\mathsf{pk} \in \text{multi-signer-set}] \Big) \geq t\ \{\mathtt{return\ 1}\}\ \mathtt{else}\ \{\mathtt{return\ 0}\}$$

3.7.4 A light-client validation protocol

The most trustless version of the validation protocol is undeniably based on light-clients. A light-client is a system that provably tracks the consensus of a certain blockchain and exposes an API to query the state of that blockchain with a cryptographic proof. In the context of a bridge, light-clients are useful for proving state updates of the bridged blockchain on either side of the bridge. If we assume the existence of light-clients for each blockchain on our bridge then we can prove that the storage of an Anchor System instance has updated.

- `validate`(message: Message, proof: Vec<u8>) $\longrightarrow \{0, 1\}$

1. Using a light-client $\mathcal{L}$, we simply verify the proof:

$$\mathtt{if}\ \mathcal{L}.\mathsf{verify\_state\_proof}(\mathsf{message}, \mathsf{proof})\ \{\mathtt{return\ 1}\}\ \mathtt{else}\ \{\mathtt{return\ 0}\}$$

4 The DKG Protocol

The Webb Protocol is built around a more robust validation mechanism than what is described in the examples above, called the DKG Protocol. The main problem with the example validation mechanisms lie in their lack of flexibility. For a single signer, the control over message validation is centralised and prone to exploits. Key management is a hard problem and so using a single key to centralise validation is not sufficient in the grand scheme of building decentralised networks. The benefit is that storage is minimal and the runtime complexity is minimized to a single elliptic-curve public key recovery for message validation.

Naturally, we may think a multi-signer system is better because it allows us to decentralise the set over an arbitrarily large threshold t. While this is better for decentralizing control over message validation, as t grows, our storage and runtime complexity grows, since we must maintain all of these keys everywhere we process messages and verify signatures for them. In a network of many anchors, this doesn't scale as well as we would like; each anchor must store all the keys, the threshold t, and validate t signatures on each message received. Moreover, changing the set of signers requires changing them everywhere, which incurs additional overhead over properly decentralizing the validation mechanism and building a permissionless, dynamic set of validators.

We want the benefit of both worlds. We want the low storage and runtime complexity of a single-signer solution with the decentralization benefits of a multi-party system. For this, we will use a distributed key generation and threshold signing system.

4.1 Distributed key generation

A distributed key generation protocol is a multi-party protocol in which n parties communicate amongst each other to generate a shared public and private keypair. For our purposes, we are interested in distributed key generation protocols that can be used for generating signatures, often referred to as t-threshold signatures. t-Threshold signatures are a type of digital signature that can only be created if a threshold $t + 1$ of parties participate in the signing protocol honestly. To that, we formalise this protocol, adapted from the definitions in [1].

Definition 6 ***Signature schemes***. *A signature scheme $\mathcal{S}$ is an efficient three-part protocol consistenting of a key-generation, signing, and verify functions:*

1. $(\mathsf{pk},\mathsf{sk}) \longleftarrow \mathsf{KeyGen}(\lambda)$ - *The key generation protocol takes as input a security parameter λ and outputs a public verification key and private signing key pair.*

2. $\sigma \longleftarrow \mathsf{Sign}(m,\ \mathsf{sk})$ - *The sign protocol takes as input a message m and a private signing key* sk *and outputs a signature σ. The algorithm can be randomized so there may exist many valid signatures.*

3. $\{0,1\} \longleftarrow \mathsf{Verify}(\mathsf{pk},\ m,\ \sigma)$ - *The verify function outputs a boolean bit indicating if the signature σ of message m is a valid signature under the public key* pk.

Definition 7 ***(t,n)*-threshold signature schemes**. *A (t,n)-threshold signature scheme $\mathcal{TS}$ is a signature scheme $\mathcal{S}$ distributing signing among **n** parties where any **t+1** of the parties can collaboratively create a digital signature. A threshold signature σ from $\mathcal{TS}$ is also a valid signature in $\mathcal{S}$. The scheme consists of the following protocols.*

- $(\mathsf{pk},\{\mathsf{sk}_i\}_{i\leq n}) \longleftarrow \mathsf{DistKeyGen}(\lambda)$ - *The distributed key generation protocol takes as input a security parameter λ and outputs a group public key* pk *as well as a secret share* sk_i *for each party $i \leq n$. Each party i only receives their share* sk_i *and none other. Everyone receives* pk.

- $\sigma_i \longleftarrow \mathsf{TresholdSign}(m,\ \mathsf{sk}_i)$ - *The threshold signing protocol takes as input a message m, a secret share* sk_i *for party i, and outputs a signature σ_i.*

- $\sigma \longleftarrow \mathsf{Aggregate}(\{\sigma_{i_j}\}_{j \le t+1})$ - *The aggregate protocol takes any $t+1$ signature shares $\sigma_{i_j},\ i_j \le n$ over a common message m and aggregates them into a single digital signature σ such that the recovered public key for σ is* pk.

Consider an instantiation of a distributed key generation protocol within the greater context of a validation protocol. We start with n parties and a threshold $t < n$. We assume successful key generation for the lifecycle of future signing operations. Therefore, we have a single public key pk that we will store and replicate everywhere messages are processed.

4.1.1 A (t, n)-threshold-signing ECDSA validation protocol.

In the (t, n)-threshold signing ECDSA validation protocol, a (t, n)-threshold ECDSA public key dictates the mechanism. That is, any message signed and aggregated by $t + 1$ parties' key shares will output 1 and 0 otherwise. The group public key pk must be replicated everywhere messages are relayed and processed.

- `validate`(message: Message, proof: EcdsaSignature) $\longrightarrow \{0, 1\}$

  1. The message is hashed using the `keccak_256` hash function, yielding `keccak_256`(message).
  2. Using an ECDSA elliptic-curve public key recovery algorithm, denoted `ecrecover`, we recover the uncompressed ECDSA public key.

$$\mathsf{public\_key} = \texttt{ecrecover}(\mathsf{proof}, \texttt{keccak\_256}(\mathsf{message}))$$

  3. Lastly, we verify the equality against the group public key.

$$\texttt{if}\ \mathsf{public\_key} = \mathsf{pk}\ \{\texttt{return 1}\}\ \texttt{else}\ \{\texttt{return 0}\}$$

The example above looks and feels more similar to the single-signer case, yet it decentralises control over multiple parties. In essence, we've abstracted the complexity of a multi-party scheme to an auxiliary distributed protocol.

4.2 Key Rotation Protocol

A truly decentralized validation mechanism must also allow the set of authorities participating in the protocol to change over time. Old members may want to leave and new members may want to join. In order to support such functionality, we introduce the Key Rotation Protocol. We assume the that the Key Rotation Protocol has access to an authority set selection system that updates authority sets at pre-defined time intervals. This system updates the current and next set of authorities on each session rotation. We use $\mathbf{a}_i$ to denote the authority set of session i. Let $\mathsf{sk}_{\mathbf{a}_i} = \{\mathsf{sk}_a\}_{a \in \mathbf{a}_i}$ be the secret key shares of authorities in set $\mathbf{a}_i$.

The Key Rotation Protocol runs in sessions of length L and executes a multi-party protocol with initial threshold $t < |\mathbf{a}_i|,\ i \in \{0, 1\}$:

1. On session $i = 0$, the current authority set $\mathbf{a}_0$ executes a t-threshold-signature schemes

$$\mathsf{pk}_{\mathbf{a}_0} \longleftarrow \mathsf{DistKeyGen}(\lambda, t, |\mathbf{a}_0|)$$

2. On session $i \geq 0$, after a time length L has elapsed from the beginning of i:
    (a) The authority set selection system outputs a candidate *next* authority set $\mathbf{a}_{i+1}$
    (b) The authorities $\mathbf{a}_{i+1}$ execute

$$\mathsf{pk}_{\mathbf{a}_{i+1}} \longleftarrow \mathsf{DistKeyGen}(\lambda, t, |\mathbf{a}_{i+1}|)$$

3. On session $i \geq 0$, if $\mathsf{pk}_{\mathbf{a}_{i+1}}$ succeeds to generate, pick a subset $S \subseteq \mathbf{a}_i,\ |S| = t + 1$ of signing parties and execute

$$\begin{aligned} m &\longleftarrow \mathsf{concat}(i+1,\ \mathsf{pk}_{\mathbf{a}_{i+1}}) \\ \sigma &\longleftarrow \mathsf{Aggregate}(\{\mathsf{ThresholdSign}(m, \mathsf{sk})\}_{(\mathsf{pk},\mathsf{sk}) \in S}) \end{aligned}$$

   where concat computes the concatenation of the hex byte representations of the arguments.

4. On session $i \geq 0$, if m, σ succeed to generate, execute

$$b \longleftarrow \mathsf{Verify}(\mathsf{pk}, m, \sigma)$$

    (a) If $b = 1$:
        i. Execute the key rotation, setting the new global public key to $\mathsf{pk}_{\mathbf{a}_{i+1}}$ and current authorities to $\mathbf{a}_{i+1}$
        ii. End the current session and start session $i + 1$
        iii. Return to (2)
    (b) If $b = 0$ repeat (3).

5. On session $i \geq 0$, if $\mathsf{pk}_{\mathbf{a}_{i+1}}$ fails to generate after a timeout $\mathsf{KEYGEN\_TIMEOUT}$:
    (a) Start or increment a retry counter c and repeat from (2b)
    (b) If $c == \mathsf{RETRY\_LIMIT}$
    (c) Execute the ***Misbehaviour Protocol*** for key generation defined in the next section and repeat from (2a)

6. On session $i \geq 0$, if σ fails to generate and/or verify successfully after a timeout $\mathsf{SIGN\_TIMEOUT}$:
    (a) Execute the ***Misbehaviour Protocol*** for signing defined in the next section
    (b) Repeat from (3)

```
trait IProto<P, S, N> {
    fn get_session(&self) -> N
    fn get_session_start_block(&self) -> N
    fn get_public_key(&self) -> P;

    fn set_session(&self, index: N);
    fn set_public_key(&self, pk: P);

    fn verify<P, S>(pk: P, msg: Vec<u8>, sig: S) -> bool;
    async fn execute_keygen(&self) -> Result<(), Error>;
    async fn threshold_sign(&self) -> Result<S, Error>;
    async fn wait_for_sigs(&self, s: S) -> Result<Vec<S>, Error>;
    async fn aggregate(&self, sigs: Vec<S>) -> Result<S, Error>;

    async fn sign_next_key(&self, key: P) -> Result<S, Error> {
        let next_session = self.get_session() + 1;
        let mut msg = Vec::new();
        msg.extend_from_slice(next_session.to_bytes());
        msg.extend_from_slice(key.to_bytes());

        let sig_share = self.threshold_sign(msg).await?;
        let all_shares = self.wait_for_sigs(sig_share).await?;
        self.aggregate(all_shares).await?
    }

    async fn rotate_session(&self) -> Result<(), Error> {
        let start = self.get_session_start_block();
        if curr_session.start_block + L <= now {
            let next_key = self.execute_keygen().await?;
            let msg = self.sign_next_key(next_key).await?;
            if verify(self.get_public_key(), msg, sig) {
                self.set_session(curr_session + 1);
                self.set_public_key(next_key);
            }
        }
    }
}
```

Figure 2: Pseudocode for a Key Rotation Protocol implementation

4.3 Misbehaviour & Reputation Protocol

Multi-party protocols are not guaranteed to succeed; machines may fail, act maliciously, and simply stop sending messages to peers. Therefore a misbehaviour reporting protocol is necessary for identifying and sharing information on peer misbehaviours. The Misbehaviour Protocol we describe tracks a reputation for each peer and selects participants for the Key Rotation Protocol and potentially other protocols down the line using their latest reputation. These reputations dictate who participates and executes, successfully, any of the algorithms in $\mathcal{TS}$ as part of the multi-party computation.

The misbehavior protocol runs as a threshold voting protocol for all non-verifiable misbehaviours. A non-verifiable misbehaviour is a misbehaviour that cannot be verified on-chain using a fraud proof. An example of a non-verifiable misbehaviour is a timeout of a node, which can't trustlessly be proven since nodes can be DDOS'ed or fail erratically.

We consider two classes of non-verifiable misbehaviours due to timeouts: key generation and sign timeouts. We use the signature threshold t to define the minimum required votes necessary to successfully report a misbehaving peer for sign timeouts and $t + 1$ to report a misbehaving peer for key generation timeouts. Once these amounts are submitted, the system decrements the misbehaving peer's reputation and jails them from participating in that stage of the protocol for a jail time.

The reputation function $\mathsf{reputation}_t$ is a bounded function that yields the reputation of a participant i. The update function is defined by a decay factor $\alpha \in [0, 1)$. A participant's reputation follows the following update rule:

1. On a successful action:
$$\mathsf{reputation(i)}_{t+1} = \mathsf{reputation(i)}_t * \alpha + 1$$
2. On a misbehaviour report:
$$\mathsf{reputation(i)}_{t+1} = \mathsf{reputation(i)}_t * \alpha$$

From this, the reputation is bounded by $\frac{1}{1-\alpha}$.

4.4 Validator and Authority Selection

The protocol is geared towards a Proof of Stake or Proof of Authority based setting. For the purposes of the paper we will restrict our attention to Proof of Stake. Thus, validators of the network are selected using a Proof of Stake mechanism, i.e. token-weighted validator selection. Let there be k validators or nodes participating in this protocol. We consider a (t, n)-threshold signature DKG with $t < n \leq k$.

We refer to the authorities that run the DKG as the best authorities. The best authorities are simply those chosen by highest reputation. If $n < k$, we use the reputation system to select the best n authorities out of the aggregate validator set.

The selection mechanism proceeds as follows:

1. Each session, a new validator set of size k is selected.
2. Out of the k validators, we select the n best authorities to run the DKG.
3. (t, n, k) are governable parameters and updates take effect after each session rotation.

5 The Webb Protocol

The combined Anchor and Validation protocol lays the foundation for the Webb Protocol. For specificity we will describe the overall system using the DKG Protocol.

5.1 Protocol Sketch

Consider a Proof of Stake (PoS) blockchain protocol where L blocks constitutes a session period. At the beginning of each session from the genesis session, the current and next validator sets $\mathbf{a}_i, \mathbf{a}_{i+1},\ i \geq 0$ are selected using the underlying Proof of Stake election mechanism.

We will use the validator set to bootstrap a DKG Protocol from the start. That is, we will execute the Key Rotation Protocol each session and Misbehaviour Protocol throughout the blockchain's execution. On this blockchain, we will also govern a set of parameters that control various aspects of the protocol. This can be implemented as a smart contract or a core primitive of the system. Specifically, we will govern:

1. The validators who can participate in both consensus and the DKG, using PoS and reputations.
2. The thresholds t, n for the t-out-of-n threshold signature protocol for the current, next, and following session.

Note: If the validator selection mechanism selects validator sets that conflict with the values of t, n then those values are updated to be compatible with the validator set automatically. For example, if the size of the set is smaller than n, the protocol will change n to be the validator set size.

5.1.1 Sketch

After a period of L blocks has elapsed in each session, the n_{next} next best authorities are selected by reputation to participate in the $(t_{\mathsf{next}}, n_{\mathsf{next}})$-distributed key generation for the next session. From here, we leverage the authorities, non-exclusively, of the blockchain to act as relayers over Anchor System instances using a combination of the threshold and light-client based validation protocol. We say non-exclusive to indicate that anyone can fulfill the role of a relayer. A protocol sketch for a connected set of anchors is as follows:

1. Relayers listen for merkle tree insertions into those anchors
2. Relayers relay messages to the Webb blockchain for light-client verification and subsequently threshold-signing.
3. Authorities generate threshold-signatures if the light-client message validation succeeds.
4. After successfully generating threshold signatures of these merkle tree updates, relayers relay and submit transactions of this signed event back to each neighboring anchor.
5. Anchors modify their internal state for the neighboring anchor being updated.
6. Users can now generate valid cross-chain zero-knowledge proofs of membership for newly inserted data and apply them on their target destination.

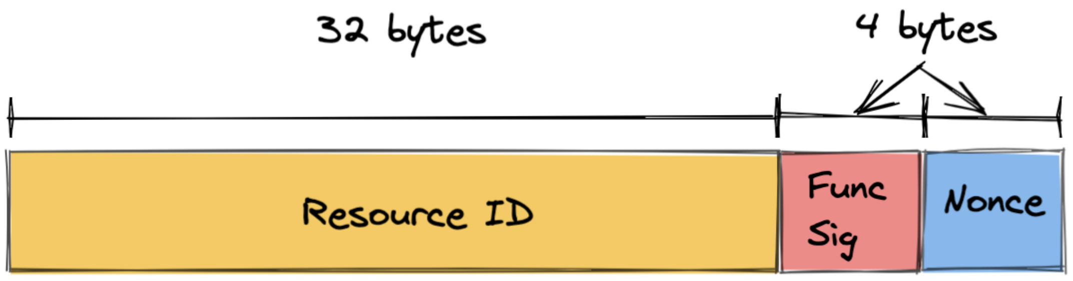

Figure 3: A message/proposal header

5.2 Message Types

The protocol utilizes formatted messages to transfer information between the protocol instances and the DKG protocol. These messages are constructed and adhere to a standard structure.

Definition 8 *A **resource identifier (resource ID)** is a unique 32-byte identifier that contains information about a resource (such as a smart contract) and a blockchain identifier of where the resource exists (such as a unique EVM chain ID).*

We use resource identifiers to identify smart contracts or indices of core runtime functionality. The motivation is to have a unique identifier that also differentiates between the blockchain where these resources are deployed to. Resource identifiers prevent replay attacks in this manner, since we can add validation on the resources themselves to ensure the the message being executed is targeting such a resource on the correct blockchain.

Each message contains a message or proposal header that describes the target execution environment such as Ethereum. If a message is meant to be processed on a smart contract on Ethereum, the message header would contain identifying information about this contract and its underlying environment.

Definition 9 *A **message header** (**proposal header**) is a 40-byte prefix attached to all messages used in the Webb Protocol. It is the concatenation of:*

- *32-bytes for an executing resource identifier.*
- *4-bytes for an executing function identifier.*
- *4-bytes for a message nonce.*

A diagram of the message header is provided in 3

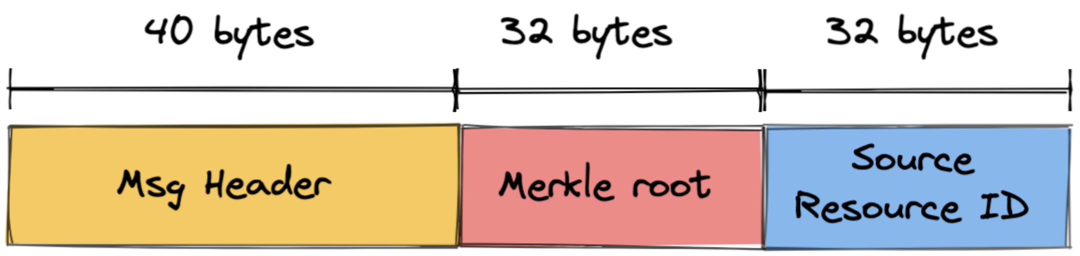

Figure 4: An anchor update proposal message

5.2.1 Anchor Update Message

The anchor update message is the primary message used to update the edges of connected anchors. The message contextualizes source and target identifying information about where an event occurred and where it is meant to be executed. For a given anchor update, a unique message is proposed to each connected neighbor using a different target resource identifier.

Definition 10 *The* ***anchor update message*** *is a 104-byte message that links a source anchor with a target anchor using their resource identifiers. It is the concatenation of:*

- *40-bytes for the proposal header containing the execution anchor resource ID.*
- *32-bytes for a new merkle root of the source chain anchor.*
- *32-bytes for the source anchor resource ID.*

A diagram of the anchor update proposal message is provided in 4

We relay these messages to the DKG for signatures and then we relay the signature and message pairs to the target anchor that is connected to the source anchor where the update has occurred.

5.3 Hybrid threshold / light-client validation protocol

The Webb Protocol will initially utilize a hybrid threshold and light-client validation protocol before realizing the fully trustless light-client based version. This is due to the reality that building bi-directional light clients is a monumental effort both in research and engineering. We describe our current research in the Appendix.

By hybrid threshold and light-client validation, we mean explicitly that:

1. Messages that are signed with a threshold signature are considered valid.
2. Messages should only be signed (enforced in protocol) if they are proven to be true against a light-client $\mathcal{L}$.

5.3.1 Light-client message validation

Our main goal in utilizing light-client here is to provide trustlessness on the messages being signed. Additionally, it prepares the protocol for a future migration towards fully light-client based bridging. For now, we are minimizing the trust surface for bridges built using this architecture by some order of magnitude since we can trustlessly reason about cross-chain state on the Webb blockchain and enforce crypto-economically that this state gets signed. We are only relying on the DKG authorities to sign messages generated from verifiable state across our Anchor System instances. The feedback loop is closed and allows for slashing DKG authorities in the event they sign invalid data. This is done simply by providing a signature and proving there was no submitted verifiable state used to construct it.

The light-client message validation mechansim trustlessly verifies the validity of messages proposed to the Webb blockchain about Anchor System instances on other blockchains. For simplicity, we restrict our attention to blockchains that have *finality* for the Anchor System instances.

For each blockchain being bridged consider its light client $\mathcal{L}$ on the Webb blockchain, we maintain:

1. The latest finalized header.
2. The logic for verifying and updating the light client's latest finalized header.
3. The logic for verifying state and events against the finalized header.

Using $\mathcal{L}$ and the functionality above, we can now:

1. Prove the state of an Anchor System instance has updated.
2. Generate the corresponding state proof.
3. Submit and verify it on the Webb blockchain.
4. If successful, generate an **AnchorUpdateMessage** and submit it for threshold signature generation.

5.4 Proposal Pipeline

The Webb Blockchain can be viewed as a generic signing service for certain proposal payloads that pass validation criteria. We can generalize this to support arbitrary proposals over verifiable data on any of the chains as well and leave this for the Appendix.

In this light, the Webb Blockchain also acts as a cross-chain governance system for controlling cross-chain applications. Since it can generate signatures, the Webb blockchain can trigger application updates based on valid signature verifications. The lifecycle of proposals is then defined by these processes for proposal creation, signing, and submission.

5.4.1 Proposal Creation

Proposals are created through 2 main flows:

1. Light-client proofs of data.
2. Token-voted proposals.

The former flow we have already described above; certain proposals such as **AnchorUpdateMessage** proposals are created on-chain if there exists a submitted proof of the underlying storage state of the Anchor System instance. In the future, arbitrary payloads will be supported if valid proofs of existence are presented.

The latter flow defines the method in which the token holders of this Webb blockchain can create proposals for the underlying threshold signature system to sign. This can follow any underlying token weighted voting system such as quadratic voting, conviction voting, majority voting, and more. Proposals created by token holders allow us to jump-start the bridging and governance for the applications deployed on a Webb blockchain.

Example 4 *Consider two Anchor System instances on different blockchains $\mathcal{A}$ and $\mathcal{B}$. Before these instances are bridged together, relayers will have no idea where to relay the signed updates to their underlying merkle trees. The system's bridging is left as a task for the token holders of the Webb blockchain, to create the necessary **AnchorUpdateMessage** proposal that initially connects the Anchor System instances on $\mathcal{A}$ with $\mathcal{B}$ and vice versa.*

5.4.2 Proposal Signing and Submission

Once proposals are created in a Webb blockchain, they are added to an unsigned proposal queue. The current DKG authorities listen for changes to the unsigned proposal queue and begin to execute the threshold signature MPC protocol. Since unsigned proposals are on-chain at this point, failure to sign these proposals can penalized through any developer-specified slashing conditions.

The protocol proceeds through this process ad infinitum:

1. Listen for insertions into the unsigned proposal queue.
2. Initiate a threhsold signing protocol over new unsigned proposals.
3. Upon successful threshold signature generation, submit this signature back on-chain.
4. Clear the unsigned proposal from the unsigned proposal queue upon successful on-chain submission.

Signed proposals are persisted on-chain for an indefinite period of time. Any user or relaying service can watch for updates to the signed proposals storage system on the Webb blockchain under question. With these signatures, these same entities can submit state changing transactions to the Anchor System instances that rely on the *(t,n)*-threshold signature validation mechanism with this blockchain's threshold distributed key as the governing key.

5.5 Open-source Software Implementations

The implementations of the blockchain and Anchor Systems are open source and built in a variety of blockchain ecosystems.

- The DKG Protocol is implemented as a Substrate based blockchain
- The EVM based Anchor System is implemented in Solidity
- The Substrate based Anchor System is implemented in Substrate

References

[1] Rosario Gennaro and Steven Goldfeder. "One round threshold ECDSA with identifiable abort". In: *Cryptology ePrint Archive* (2020).

[2] Kobi Gurkan, Koh Wei Jie, and Barry Whitehat. "Community proposal: Semaphore: Zero-knowledge signaling on ethereum". In: *Accessed: Jul* 1 (2020), p. 2021.

[3] Gavin Wood et al. "Ethereum: A secure decentralised generalised transaction ledger". In: *Ethereum project yellow paper* 151.2014 (2014), pp. 1–32.