Community Proposal: Semaphore: Zero-Knowledge Signaling on Ethereum

Kobi Gurkan
Ethereum Foundation and cLabs

Koh Wei Jie
Ethereum Foundation

Barry Whitehat
Independent

March 31, 2020

Abstract

Privacy has been a big concern in the blockchain space. While different specific solutions have been introduced to introduce more privacy into systems, they remain focused on specific problems and are complex to extend and deploy.

We introduce Semaphore - a framework for zero-knowledge signaling on Ethereum. It allows a user to broadcast their support of an arbitrary string, without revealing who they are to anyone, besides being approved to do so. Semaphore is meant to be used a base layer for signaling-based applications - mixers, anonymous DAOs, anonymous journalism, etc.

Semaphore is designed to allow building applications in a modular fashion. Normally, they would be implemented as a smart contract that would manage the onboarding of new identities and would define the conditions for signals to be accepted for broadcast, besides passing the checks of the Semaphore layer. Being deployed on Ethereum, it allows interaction with other applications residing on the Ethereum blockchain.

We provide an efficient implementation of our framework, in the form of two example applications - a mixer and an anonymous survey dApp. Our implementation is built in a way that makes it flexible to extend and clear to deploy.

Contents

1 Introduction

Privacy has been a big concern in the blockchain space. Financial privacy solutions has been introduced in the form of systems based on mixing [9, 21], systems based on decoys, such as Monero [15], and systems based on zkSNARKs, such as Zcash [5]. Mixing-based and decoy-based solutions provide a limited anonymity set per transaction, while solutions based on zkSNARKs provide an always growing anonymity set. Even though, zkSNARKs-based financial privacy has been mainly possible up until 2019 in the Zcash blockchain, which is its own separate blockchain, making interoperability with other applications complex and error-prone.

Financial privacy is not the only kind of privacy desired in the blockchain space. Other areas of interest include anonymous authentication, where members of a group can login to a service without revealing which member of the group they are and in the process hide their transaction history, and anonymous DAOs (Decentralized Autonomous Organizations), where users can anonymously take actions (e.g., vote) without having their individual actions linked to each other.

Since the introduction of EIP 196 [11] and EIP 197 [12], the costs for verifying zkSNARK proofs on Ethereum have been practical, and even more so after EIP 1108 [10]. This opens the door to deploy zkSNARK-based privacy solutions in a way that interacts with other deployed applications on the same blockchain, where in Ethereum's case are general-purpose smart contracts.

We introduce Semaphore, a zero-knowledge signaling framework highly inspired by Zcash, providing a method for users who are part of a group to broadcast an arbitrary string without exposing their identity. Semaphore is designed as a base layer that can be used to build different applications, which we motivate in this document. During the design of Semaphore, we choose cryptographic primitives that strongly protect the users long-term anonymity, while for efficiency we use newer primitives that are now receiving more attention [14]. Moreover, Semaphore is designed to support external authorization, such as hardware wallets, to allow safer signaling, by the user of signatures inside the circuit.

Semaphore is designed to be efficient to run both in a browser, on mobile and on Ethereum's EVM, giving rise to a system that is practical for every-day uses. We provide an implementation of both the zkSNARKs and contracts of Semaphore an an example application, MicroMix, which is built on top of Semaphore.

We note that Semaphore can be seen as built from gadgets that have been discussed in [22], as its components are membership proofs, commitments, signature verification and a PRF. The gadgets listed in [22] can replace these parts in Semaphore, depending on the desired efficiency and security parameters.

In summary, Semaphore is a generic zero-knowledge signaling protocol, with a concrete efficient and reasonbly secure instantiation on Ethereum, coupled with an easy-to-use and easy-to-extend implementation.

2 Notation

$\mathbb{b}$ means the type of bit values, i.e. $\{0, 1\}$. $\mathbb{B}$ means the type of byte values, i.e. $\{0..255\}$.

$\mathbb{b}^{[\ell]}$ means the set of sequences of ℓ bits, and $\mathbb{B}^{[k]}$ means the set of sequences of k bytes.

$l_{\texttt{MerkleDepth}}, l_{\texttt{MerkleNode}}, l_{\texttt{Position}}, l_{\texttt{ExternalNullifier}}, l_{\texttt{NullifierS}}, l_{\texttt{BabyJubjubCapacity}}, l_{\texttt{BabyJubjubBits}}, l_{\texttt{BNCapacity}}, l_{\texttt{BNBits}}, l_{\texttt{BabyJubjubCofactor}}, l_{\texttt{EthereumAddress}}$ are constants that will be instantiated in the concrete protocol description.

pad32 means padding a byte sequence with bytes of value 0 to the left, until reaching a 32-byte array.

BN254, BabyJubjub, $\mathbb{F}_q, \mathbb{F}_r, \mathbb{F}_s, \mathbb{J}$ are defined in section 5.2.

$\mathsf{NumToBits}_x$ converts a number to its little-endian bit representation. NumToBitsStrict does the same, with verification that the bit representation is the one that generates the number with value under $\mathbb{F}_s$, preventing aliasing.

`uint256` is a 32-byte integer.

MedDSA is defined in 4.4 and instantiated in 5.5.

$\mathsf{Project}_{\mathsf{x}}$ means taking the x coordinate of an elliptic curve point.

3 Concepts

3.1 High-level design

3.1.1 Overview

Semaphore allows users in a group to anonymously broadcast a signal on different topics without exposing their identities. These signals are arbitrary messages. Topics are later on called *external nullifier*s. The main security claims of Semaphore are:

1. Users who broadcast a signal will not expose their identity. Specifically, an adversary will only know they're a user in the group, but not which user.
2. Users cannot broadcast two different signals on the same topic twice.

Semaphore-based systems have to define the following rules when deployed:

1. Which users are allowed to join the group
2. What happens when a user successfully broadcasts a signal

These rules are usually specified as part of a smart contract's code when deployed. For concrete examples of such rules, see 6 and 7.

Whenever a user joins the group, a commitment to their identity details is added to an incremental Merkle tree.

In order to preserve anonymity of the broadcaster, Semaphore relies on a zk-SNARK proof that shows that the user identity's commitment is a member of the set by showing that its a leaf in the tree. Additionally, it exposes a unique string called a *nullifier* which is derived from the identity details and the topic. The smart contract verifies a broadcast nullifier is unique, thus preventing more than one broadcast per identity and topic.

Additionally, Semaphore requires that the identity owner provides a signature by the identity's public key as part of the proof. This separation allows moving the final authorization of a broadcast to a hardware wallet, making it harder for an attacker to perform a successful broadcast. Even if an attacker compromises the user's machine generating the proof, containing the user's specific identity data, the attacker can only reasonably deanonymize the user, but cannot fraudulently broadcast a signal.

3.1.2 Choice of cryptographic primitives

We use 3 different hash functions throughout Semaphore. While this introduces complexity, it serves the following purposes:

1. PedersenHash is used to generate identity commitments. We use it for this purpose since it can become a hiding commitment by introducing a random element controlled by the user.
2. Blake2s is used to generate nullifiers. We use it for this purpose since it is widely considered as a good cryptographic hash function, that can be used in place of a random oracle. Nullifiers require a deterministic hash function whose outputs can't reasonably be inverted. It's the heaviest of the hash functions we use and so we use it sparingly.
3. MiMCSponge is used to build the identity Merkle tree. It's an algebraic and SNARK-friendly hash function that can be used in place of a random oracle. While no severe attacks are known, it's still novel and we feel more time is required to analyze it by the wider community. Breaking MiMCSponge in Semaphore will only result in being able to prove you're a member of the group - it will not break past anonymity (because of the

use of Blake2s and will not reveal identity details because of the use of PedersenHash.

3.2 Identity keys

Identities in Semaphore must generate an MedDSA private key $\mathsf{id}_{\mathsf{key}}$, which will be used to authenticate their signals to the system.

3.3 Identities

Users registering to the system possess an *identity* they must keep secret. A commitment to this structure is known to the public.

An *identity* is a tuple $(\mathsf{id}_{\mathsf{pub}}, \mathsf{id}_{\mathsf{nullifier}}, \mathsf{id}_{\mathsf{trapdoor}})$, where:

- $\mathsf{id}_{\mathsf{pub}}$: MedDSA.Public is the public key corresponding to the identity's private key $\mathsf{id}_{\mathsf{key}}$.
- $\mathsf{id}_{\mathsf{nullifier}} : \mathbb{B}^{[31]}$ is a random sequence of bytes.
- $\mathsf{id}_{\mathsf{trapdoor}} : \mathbb{B}^{[31]}$ is a random *commitment trapdoor*.

An *identity* is generated locally by a user registering to Semaphore, where $\mathsf{id}_{\mathsf{key}}$ and $\mathsf{id}_{\mathsf{nullifier}}$ are sampled randomly. Both $\mathsf{id}_{\mathsf{pub}}$ and $\mathsf{id}_{\mathsf{nullifier}}$ are not revealed publicly, and are used by the *zero-knowledge proof* to check the *identity commitment* exists in the *identity commitment tree* and the *nullifier map* is checked so that the *nullifiers hash* corresponding to the *identity nullifier* was not used to broadcast a signal for the same *external nullifier*.

An *identity commitment* on an *identity* $\mathbf{id} = (\mathsf{id}_{\mathsf{pub}}, \mathsf{id}_{\mathsf{nullifier}}, \mathsf{id}_{\mathsf{trapdoor}})$ is computed as

$$\mathsf{Commitment}(\mathbf{id}) = \mathsf{Commit}_{\mathsf{id}_{\mathsf{trapdoor}}}(\mathsf{id}_{\mathsf{pub}}, \mathsf{id}_{\mathsf{nullifier}})$$

where Commit is a Pedersen hash instantiated in 5.3.2.

3.4 External Nullifier

The *external nullifier* is combined with the *identity nullifier* to form an opaque nonce value, which uniquely identifies a signal from an identity for a specific *external nullifier*.

3.5 Identity Commitment Tree

Users normally register identities by broadcasting a transaction containing an *identity commitment* to the application smart contract, where custom valida-

tion rules are checked, depending on the use-case - see 6 for an example. The Semaphore smart contract then adds the identity to the *identity commitment tree*.

Specifically, the smart contract maintains a $log(n)$ amount of tree nodes, including the tree root, allowing to efficiently insert new tree leaves. Additionally, the smart contract stores all the leaves, so users could rebuild the tree locally when wishing to find the path from their leaf to the root, required by the *zero-knowledge proof*.

The *identity commitment tree* is stored in the state.

3.6 Nullifier Map

The *nullifier map* is a hash map maintained by Semaphore, which prevents double signaling by the same identity with the same *external nullifier*. Each time a signal is broadcast, it includes a *nullifiers hash* derived from the *identity commitment* position in the tree, the *identity nullifier* and the *external nullifier*. This *nullifiers hash* is checked for existence in the hash map, preventing multiple signals by the same identity for the same *external nullifier*.

The *nullifier map* is stored in the state.

3.7 Signal map

The *signal map* is a hash map maintained by Semaphore, which stores all the signals successfully broadcast by users. This serves as a source for users to read historic signals.

The *signal map* is stored in the state.

3.8 Relayer

Ethereum transactions are tied to a specific address, and transactions sent from this address consume Ether to pay for gas. When a signal is broadcast, the Ethereum address used for the transaction might be used to break the anonymity of the broadcaster. Semaphore allows signals to be received from any address, allowing a *relayer* to broadcast a *signal* on behalf of a user. Applications might provide rewards for relayers and implement front-running prevention mechanisms, such as requiring the signals to include the relayer's address, binding the signal to that specific address.

3.9 Relayer Registry

We decentralise transaction abstraction via a burn relay registry.

A *relayer* does the following:

1. Receives transactions via an off-chain API
2. Simulates said transactions to see if their balance increases if executed (this implies that the target contract will transfer ETH or tokens to msg.sender)
3. Execute the transaction and burn a fraction of the fee earned.
4. The fee is both a reputation and anti-spam mechanism

There is a *relayer registry* contract which keeps track of relayers' ETH addresses, the number of transactions relayed, and the total amount of ETH burned.

4 Abstract Protocol

4.1 Hash functions

$\mathsf{MerkleCRH} : \mathbb{F}_r \bigtimes \mathbb{F}_r \to \mathbb{F}_r$ is the hash function used in the Merkle tree.

Security requirements: MerkleCRH must be collision-resistant.

Design rationale: MerkleCRH's collision-resistance ensure that the Merkle root generated from a leaf set is unique, and that it is infeasible to find a leaf that is not in the set and can be proven to be included in it.

4.2 Pseudo random functions

$\mathsf{NullifiersPRF}_x(y, z) : \mathbb{B}^{[l_{\texttt{NullifierS}}]} \bigtimes \mathbb{B}^{[l_{\texttt{ExternalNullifier}}]} \bigtimes \mathbb{B}^{[l_{\texttt{Position}}]} \to \mathbb{b}^{[l_{\texttt{BabyJubjubCapacity}}]}$ is used to derive a unique byte-string out of the identity's position in the tree, the *identity nullifier* and the *external nullifier*. The first argument is the key.

Security requirements:

1. Security definitions for Pseudo Random Functions are given in [4].
2. In addition to being Pseudo Random Functions, it is required that NullifiersPRF is collision-resistant across all x — i.e. finding $(x, y, z) \neq (x', y', z')$ such that $\mathsf{NullifiersPRF}_x(y, z) = \mathsf{NullifiersPRF}_{x'}(y', z')$ should not be feasible.

Design rationale: NullifiersPRF is meant to preserve the anonymity of the broadcaster by preventing linkability between multiple signals from the same broadcaster. Being a conservative bit-twiddling hash, it aims to provide security against adversaries wishing to deanonymize past signals, which may include quantum adversaries.

4.3 Commitments

$\mathsf{Commit} : \mathbb{B}^{[31]} \bigtimes \mathbb{J}^{(r)} \bigtimes \mathbb{B}^{[31]} \to \mathbb{F}_r$ is the commitment scheme used for committing to identities.

Security requirements:

1. Computationally binding - It is infeasible to find $\mathsf{id_{pub}}, \mathsf{id_{nullifier}}, \mathsf{id_{pub}}', \mathsf{id_{nullifier}}'$ and $\mathsf{id_{trapdoor}}, \mathsf{id_{trapdoor}}'$ such that $\mathsf{Commit}_{\mathsf{id_{trapdoor}}}(\mathsf{id_{pub}}, \mathsf{id_{nullifier}}) = \mathsf{Commit}_{\mathsf{id_{trapdoor}}'}(\mathsf{id_{pub}}', \mathsf{id_{nullifier}}')$.
2. Perfect hiding - for all $\mathsf{id_{pub}}, \mathsf{id_{nullifier}}, \mathsf{id_{pub}}', \mathsf{id_{nullifier}}'$, the distributions $\{\mathsf{Commit}_{\mathsf{id_{trapdoor}}}(x) \mid \mathsf{id_{trapdoor}} \xleftarrow{\$} \mathsf{Commit.GenTrapdoor}()\}$ and $\{\mathsf{Commit}_{\mathsf{id_{trapdoor}}}(x') \mid \mathsf{id_{trapdoor}} \xleftarrow{\$} \mathsf{Commit.GenTrapdoor}()\}$ are equal.

Design rationale: Commit generates commitments that are publicly transmitted. It's important to provide a reasonable hiding property, since some of the elements inside the commitments could possibly be used to either deanonymize the identity when broadcasting or generate unauthorized signals, if faced against a quantum adversary. It's also important that the scheme is binding, since you shouldn't be able to either use another nullifier or public key, allowing multiple signals for the same external nullifier.

4.4 Signatures

Semaphore uses MedDSA for authorizing signals, a signature scheme almost identical to EdDSA, with the difference that the hash function used for hashing the private key and the hash function used for hashing (R, A, M) are different.

MedDSA provides:

- a type of signing keys MedDSA.Private.
- a type of verifying keys MedDSA.Public.
- a type of messages MedDSA.Message.
- a type of signatures MedDSA.Signature.
- a randomized signing key generation algorithm MedDSA.GenPrivateKey : $() \xrightarrow{\$}$ MedDSA.Private.
- an injective verifying key derivation algorithm MedDSA.DerivePublic : MedDSA.Private $\to$ MedDSA.Public.
- a hash function $H_{\mathtt{key}} : \mathsf{MedDSA.Private} \to \mathbb{B}^{[2 * l_{\mathtt{BabyJubjubBits}}]}$.
- a hash function $H_{\mathtt{message}} : \mathbb{F}_r^4 \bigtimes \mathsf{MedDSA.Message} \to \mathbb{F}_s$.

- a randomized signing algorithm $\mathsf{MedDSA.Sign}_{H_{\texttt{key}}, H_{\texttt{message}}} : \mathsf{MedDSA.Private} \bigtimes \mathsf{MedDSA.Message} \xrightarrow{\$} \mathsf{MedDSA.Signature}$
- a verifying algorithm $\mathsf{MedDSA.Verify}_{H_{\texttt{key}}, H_{\texttt{message}}} : \mathsf{MedDSA.Public} \bigtimes \mathsf{MedDSA.Message} \bigtimes \mathsf{MedDSA.Signature} \to \mathbb{b}$

For a message M and a private key $sk \xleftarrow{\$} \mathsf{MedDSA.GenPrivateKey}()$, MedDSA must satisfy $sig \leftarrow \mathsf{MedDSA.Sign}(sk, M), \mathsf{MedDSA.Verify}(\mathsf{MedDSA.DerivePublic}(sk), M, sig) = 1$.

Security requirements:

1. MedDSA must be Strongly Unforgeable under (non-adaptive) Chosen Message Attack (SU-CMA), as defined for example in [6]. This allows an adversary to obtain signatures on chosen messages, and then requires it to be infeasible for the adversary to forge a previously unseen valid (message, signature) pair without access to the signing key.
2. $H_{\texttt{key}}$ is modeled as a random oracle.
3. $H_{\texttt{message}}$ must be long enough to prevent grinding attacks.

Design rationale: We've chosen to use a signature scheme here to allow the separation of authorization and anonymity, allowing for better security for authorization. The signature may be more easily generated on a hardware device, such that an attacker that can interfere with the zkSNARK proving process, still can't generate the appropriate signature to complete it.

4.5 Identity commitment tree

The *tree state* is an incremental Merkle tree, which maintains a state of size $log(n)$ called $\mathsf{TreeState.Subtrees} : \mathbb{F}_r[l_{\texttt{MerkleDepth}} + 1]$ and allows efficient insertions. It provides the following API:

1. $\mathsf{TreeState.AddCommitment} : \mathbb{F}_r \to ()$ - adds a commitment to the right, replacing the next empty leaf. When a subtree of level i is filled, the element $\mathsf{TreeState.Subtrees}[i]$ is replaced with the root of the subtree.
2. $\mathsf{TreeState.GetLeaves} : () \to \mathbb{F}_r[2^{l_{\texttt{MerkleDepth}}}]$ - returns all the leaves stored in the tree.

Users wishing to broadcast signal must first obtain the path from the leaf of their identity commitment to a root of the *identity commitment tree*, either the current or an historic one. It's better using a recent one to prevent leaking the fact you belong to a historic and smaller anonymity set. These paths can be obtained by either:

1. Building the tree locally using the leaves at the time of signal broadcast.

2. Asking the path from a service that maintains the whole tree, at the cost of the service finding out which leaf belongs to the user.
3. Continuously maintaining a path, and incrementally updating it when leaves are added to the tree. These method is described in [5].

4.6 Nullifier map

The *nullifier map* is a hash map containing the *nullifiers hash*-es from each broadcast. This prevents multiple broadcasts of multiple signals from the same identity and the same *external nullifier*. It provides the following API:

1. NullifierMap.AddNullifiersHash : $\mathbb{B}^{[31]} \to ()$ - adds a *nullifiers hash* to the hash map.
2. NullifierMap.CheckNullifiersHash : $\mathbb{B}^{[31]} \to \mathbb{b}$ - checks if a *nullifiers hash* exists in the hash map.

4.7 Signal map

The *signal map* is a hash map containing the signals, byte strings of arbitrary length, that users have broadcast successfully. The signals are indexed by a `uint256` index. It provides the following API:

1. SignalMap.AddSignal : $\mathbb{B}^{[]} \to ()$ - stores a signal in the map.
2. SignalMap.GetSignalByIndex : `uint256` $\to \mathbb{B}^{[]}$ - retrieves a signal by an index.

4.8 Semaphore state

Semaphore's state is comprised of the *identity commitment tree*, the *nullifier map*, the *signal map* and a few other public state variables:

1. TreeState.Owner : $\mathbb{B}^{[l_{\texttt{EthereumAddress}}]}$ is the owner of the state who has permissions to perform some actions other actors cannot. It can be changed by TreeState.SetOwner, allowed only to be called by TreeState.Owner.
2. TreeState.IdentityCommitmentTree is an instance of TreeState, where addition is only allowed to be performed the owner of the state.
3. TreeState.NullifierMap is an instance of NullifierMap, where insertion is only allowed after a successful signal broadcast.
4. TreeState.SignalMap is an instance of SignalMap, where insertion is only allowed after a successful signal broadcast.

5. TreeState.ExternalNullifier : $\mathbb{B}^{[l_{\texttt{ExternalNullifier}}]}$ is the current *external nullifier*, and can be changed by the owner using TreeState.SetExternalNullifier : $\mathbb{B}^{[l_{\texttt{ExternalNullifier}}]} \to ()$.
6. TreeState.IsPermissioned : $\mathbb{b}$ determines whether signals can be broadcast by anyone or just the owner. The latter situation fits the common case where Semaphore is used as a base layer, and the application defines more complex validation checks before passing it on to Semaphore. It can be changed by the owner using TreeState.SetIsPermissioned : $\mathbb{b} \to ()$.

4.9 zkSNARK statement

Given public inputs:

- `signal_hash` : $\mathbb{F}_r$
- `external_nullifier` : $\mathbb{B}^{[l_{\texttt{ExternalNullifier}}]}$
- `root` : $\mathbb{F}_r$
- `nullifiers_hash` : $\mathbb{b}^{[l_{\texttt{BabyJubjubCapacity}}]}$

and private inputs:

- **id** = $(\mathsf{id}_{\mathsf{pub}}, \mathsf{id}_{\mathsf{nullifier}}, \mathsf{id}_{\mathsf{trapdoor}})$
- idcomm : Commit.Output
- id_path : $\mathbb{F}_r[l_{\texttt{MerkleDepth}}]$
- id_path_index : $\mathbb{b}^{[l_{\texttt{MerkleDepth}}]}$
- signature : MedDSA.Signature

the following conditions hold:

- **Identity commitment integrity** idcomm = $\mathsf{Commit}_{\mathsf{id}_{\mathsf{trapdoor}}}(\mathsf{id}_{\mathsf{pub}}, \mathsf{id}_{\mathsf{nullifier}})$.
- **Merkle path validity** (id_path, id_path_index) is a valid Merkle path from idcomm to `root`.
- **Nullifiers hash integrity** `nullifiers_hash` = $\mathsf{NullifiersPRF}_{\mathsf{id}_{\mathsf{nullifier}}}(\texttt{external\_nullifier}, \mathsf{id\_path\_index})$
- **Signal authorization** $\mathsf{MedDSA.Verify}(\mathsf{id}_{\mathsf{pub}}, (\texttt{external\_nullifier}, \texttt{signal\_hash}), \mathsf{signature}) = 1$

4.10 Groups, fields and zkSNARK proving system

We directly use the description of group operations, fields and zkSNARK proving system as described in [5], where we use the Groth16 proving system with the BN254 curve.

5 Concrete Protocol

5.1 Constants

Let:

- $l_{\texttt{MerkleDepth}} := 20$
- $l_{\texttt{BabyJubjubBits}} := 251$
- $l_{\texttt{BabyJubjubCapacity}} := l_{\texttt{BabyJubjubBits}} - 1$
- $l_{\texttt{BNBits}} := 254$
- $l_{\texttt{BNCapacity}} := l_{\texttt{BNBits}} - 1$
- $l_{\texttt{MerkleNode}} := l_{\texttt{BNBits}}$
- $l_{\texttt{EthereumAddress}} := 20$
- $l_{\texttt{BabyJubjubCofactor}} := 8$
- $l_{\texttt{ExternalNullifier}} := 29$
- $l_{\texttt{NullifierS}} := 31$
- $l_{\texttt{Position}} := 4$

$l_{\texttt{BabyJubjubCapacity}}$ and $l_{\texttt{BNCapacity}}$ are the lenghts of arbitrary bit strings that can be stored and safely be converted to a number in $\mathbb{F}_r$ without aliasing.

5.2 Groups and fields

We use the BN254 curve, whose equation is $y^2 = x^3 + 3$ over the finite field $\mathbb{F}_q$, where $q := 21888242871839275222246405745257275088696311157297823662689037894645226208583$. Its scalar field is denoted $\mathbb{F}_r$, where $r := 21888242871839275222246405745257275088548364400416034343698204186575808495617$.

Inside it, we use the embedded curve BabyJubjub, denoted $\mathbb{J}$, whose equation is $168700x^2 + y^2 = 1 + 168696x^2y^2$ over the finite field $\mathbb{F}_r$. Its scalar field is denoted $\mathbb{F}_s$.

5.3 Hash functions

5.3.1 MiMC sponge

We use the $\mathsf{MiMC-Feistel}$ permutation described in [1] instantiated over $\mathbb{F}_r^2$ with 220 rounds and 5 as the exponent, denoted P. We then define $\mathsf{MiMCSponge}(m, n)$

as a sponge construction with $\texttt{capacity} = 1, \texttt{rate} = 1$, m is the amount of input field elements and n is the amount of output field elements. $\oplus$ operations have been replaced with field additions.

We use $\mathsf{MiMCSponge}(2, 1)$ for the Merkle tree and hashing the message passing it to MedDSA.Verify and $\mathsf{MiMCSponge}(5, 1)$ in the MedDSA verifier itself.

We use the round constants generated by first hashing the string "mimcsponge" using Keccak256, and then repeatedly hashing this result using Keccak256, starting from index 1. The first and last round constants are 0, as recommended by the paper.

5.3.2 Pedersen hash and commitments

We use a Pedersen hash defined over the BabyJubjub curve, as introduced in [5]. It works by dividing an input message $M = \{M_i : \mathbb{b}\}_{i=0}^{N}$ into segments and then computing over individual 4-bit windows:

$$\mathsf{PedersenHash}(M) = \sum_{s=0}^{S-1} \sum_{w=0}^{W-1} (-1)^{M_{4w+3}} \cdot 2^{5w} (M_{4w} + 2M_{4w+1} + 4M_{4w+2}) g_s$$

where $W = 50$ and $S = \frac{N-1}{200} + 1 <= 10$. For each s, the generator g_s is computed as the first successful attempt, when incrementally trying indices from $i = 0$, finding a BabyJubjub point from a possible x coordinate calculated as $\mathsf{Blake256}(\text{"}\texttt{PedersenGenerator}\text{"} || \mathsf{pad32}(s) || \mathsf{pad32}(i))$, with the 255th bit set to 0. Blake256 is described in [3].

We define $M := \mathsf{pad32}(\mathsf{NumToBitsStrict}(\mathsf{Project_x}(l_{\texttt{BabyJubjubCofactor}} \cdot \mathsf{id_{pub}}))) \; || \; \mathsf{pad32}(\mathsf{id_{nullifier}}) \; || \; \mathsf{pad32}(\mathsf{id_{trapdoor}})$ and instantiate $\mathsf{Commit}_{\mathsf{id_{trapdoor}}}(\mathsf{id_{pub}}, \mathsf{id_{nullifier}})$ by the x coordinate of $\mathsf{PedersenHash}(M)$, where $\mathsf{id_{trapdoor}} : \mathbb{B}^{[31]}$ is a random sequence of bytes.

The reason we use $l_{\texttt{BabyJubjubCofactor}} \cdot \mathsf{id_{pub}}$ is that in signature verification we multiply by the cofactor anyway, and this allows us to uniquely identify this set of keys that pass verification by their x coordinate.

5.4 Nullifiers hash

$\mathsf{NullifiersPRF}_x(y, z)$ is defined as $\mathsf{Blake2s}(x \; || \; y \; || \; z)$, where $||$ means bitwise concatenation. Blake2s is described in [2].

5.5 Signatures

We instantiate:

- MedDSA.Public := $\mathbb{J}$.
- MedDSA.Private := $\mathbb{B}^{[32]}$.
- MedDSA.Message := $\mathbb{F}_r$.
- MedDSA.Signature := $(\mathbb{J}, \mathbb{F}_s)$.

We also instantiate $H_{\texttt{key}}$:= Blake512 and $H_{\texttt{message}}$:= MiMCSponge$(5, 1)$, where points are unpacked to their respective coordinates. Blake512 is described in [3].

5.6 Semaphore state

The Semaphore state is implemented as a smart contract on Ethereum, as can be seen in the implementation.

5.7 Implementation

Semaphore's contracts, circuits and client-side code are available on [18]. The implementation uses Circom [7] and CircomLib [8] for the circuits, Solidity for the smart contracts and its prover and verifiers are built using SnarkJS [19] and WebSnark [20].

Proofs can be created in about 15 seconds on an average browser, both on PC and mobile. Verification of proofs is done as part of the smart contract evaluation.

6 MicroMix - a Mixer

6.1 Use-case

Mixing allows users to deposit a fixed amount (e.g., 0.1 ETH) and later on withdraw that same amount to a different address that is not linked to the original depositor. This allows user to achieve greater privacy - the new address can only be claimed to be part of the set of all depositor addresses.

Semaphore allows instantiating a mixer on top of Ethereum by defining application-level rules in the form of smart contracts.

6.2 Security claims

Assuming the trusted setup was done correctly according the to code supplied:

1. Money cannot be created or destroyed.

2. Only someone who knows the private key can withdraw a leaf.
3. No double spends.

6.3 Application layer

MicroMix is built as an application on top of Semaphore. It supports mixing either ETH or ERC20. MicroMix adds important characteristics that make Semaphore suitable for a mixer use-case:

1. Identity registration happens on a deposit of a constant 0.1 ETH amount.
2. The external nullifier is constant and is the mixer's contract's address. In a mixer use-case, there shouldn't be multiple signals from the same identity, as every deposit should correspond to a single withdrawal.
3. The signal format is fixed to $\mathsf{Keccak256}(addr, relayer, fee)$, binding the withdrawal to a specific Ethereum address $addr$, to a specific relayer address $relayer$ and a fee to be paid to the relayer fee.
4. Semaphore broadcasts are premissioned, allowing the mixer contract to verify the necessary condition and pass them on to Semaphore.

6.4 Implementation

MicroMix's contracts, circuits and client-side code are available on [13].

7 OneOfUs - an anonymous survey/voting dApp

OneOfUs is an anonymous survey application, built on top of a sybil-resistance token issuance protocol, Proof of Attendence Protocol [17]. Each attendee at a conference receives a non-fungible Proof of Attendance Token (POAP). We assume that only attendees own POAP tokens associated with this event.

Attendees can register themselves to the contract as long as they own a specific event's POAP token. When they wish to respond to a question, they can use OneOfUs to generate a proof of their initial registration. This proof does not reveal their identity, but only their membership in the set of registered identities.

7.1 Security claims

Assuming the trusted setup was done correctly according the to code supplied:

1. Only users with a POAP token of a specific event can post questions.

2. Only someone who know the private key can answer a question.
3. No user can answer the same question twice.

7.2 Application layer

OneOfUs is built as an application on top of Semaphore. OneOfUs utilizes Semaphore's concepts as follows:

1. Identity registration happens on proving ownership of a POAP token.
2. The external nullifier is a hash of a question.
3. The signals are the answers to the questions.

7.3 Implementation

OneOfUs's contracts, circuits and client-side code are available on [16].

8 Acknolwedgements

We thank the Ethereum Foundation for supporting our work.

We thank the Electric Coin Company for their pioneering work and excellent technical specification, which we learned a lot from.

We thank Jordi Baylina and iden3 for their excellent zkSNARKs tools which Semaphore and MicroMix are using.

We thank Harry Roberts for many fruitful discussions.

We thank the Semaphore Society community for sharing ideas and mutual collaboration.

References

[1] Martin Albrecht et al. "MiMC: Efficient encryption and cryptographic hashing with minimal multiplicative complexity". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2016, pp. 191–219.

[2] Jean-Philippe Aumasson et al. "BLAKE2: simpler, smaller, fast as MD5". In: *International Conference on Applied Cryptography and Network Security*. Springer. 2013, pp. 119–135.

[3] Jean-Philippe Aumasson et al. "Sha-3 proposal blake". In: *Submission to NIST* 92 (2008).

[4] Mihir Bellare et al. “A concrete security treatment of symmetric encryption”. In: *Proceedings 38th Annual Symposium on Foundations of Computer Science*. IEEE. 1997, pp. 394–403.
[5] S. Bowe et al. *Zcash Protocol Specication*. 2019.
[6] Johannes Buchmann et al. “On the security of the Winternitz one-time signature scheme”. In: *International Journal of Applied Cryptography* 3.1 (2013), pp. 84–96.
[7] *Circom*. https://github.com/iden3/circom.
[8] *CircomLib*. https://github.com/iden3/circomlib.
[9] *CoinJoin*. https://en.bitcoin.it/wiki/CoinJoin.
[10] *EIP 1108*. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1108.md.
[11] *EIP 196*. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-196.md.
[12] *EIP 197*. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-197.md.
[13] *MicroMix*. https://github.com/weijiekoh/mixer.
[14] *MiMC Hash Challenge*. http://mimchash.org/.
[15] *Monero*. https://www.getmonero.org/.
[16] *OneOfUs*. https://github.com/weijiekoh/oneofus.
[17] *Proof of Attendence Protocol*. http://poap.xyz/.
[18] *Semaphore*. https://github.com/kobigurk/semaphore.
[19] *SnarkJS*. https://github.com/iden3/snarkjs.
[20] *WebSnark*. https://github.com/iden3/websnark.
[21] *ZeroLink*. https://github.com/nopara73/ZeroLink/.
[22] ZKProof. *ZKProof Community Reference. Version 0.2*. Dec. 2019. Updated versions at https://zkproof.org.