A Nullifier Protocol based on a Verifiable, Threshold OPRF

Daniel Kales[1], and Roman Walch[1]
[1] TACEO
office@taceo.io

Abstract. In this paper we describe a service capable of creating publicly verifiable nullifiers using a threshold version of a verifiable oblivious pseudo random function (OPRF). Our construction is based on the TwoHashDH OPRF, discrete logarithm equality proofs, Shamir secret sharing and zk-SNARKs.

Keywords: OPRF · VOPRF · threshold · ZKP

Contents

1 Introduction ... 2
1.1 Notation ... 3
2 Background ... 3
2.1 BabyJubJub ... 3
2.1.1 Definiton of BabyJubJub ... 4
2.1.1.1 Twisted Edwards Form ... 4
2.1.1.2 Montgomery Form ... 4
2.2 EdDSA on BabyJubJub ... 5
2.3 TwoHashDH OPRF ... 6
2.4 Mapping inputs to Curve Points ... 6
2.5 Discrete Logarithm Equality Proof ... 7
3 Our Construction ... 8
3.1 Threshold OPRF ... 8
3.2 Distributed Discrete Logarithm Equality Proof ... 8
3.2.1 Additively Shared Discrete Logarithm Equality Proof ... 8
3.2.2 Shamir Shared Discrete Logarithm Equality Proof ... 10
3.3 Full Distributed OPRF-Based Nullifier Protocol ... 11
3.3.1 Clients Zero-Knowledge Proofs ... 12
3.3.1.1 Query Proof π_1 ... 12
3.3.1.2 Nullifier Proof π_2 ... 13
3.3.2 Key Generation and Reshare ... 13
3.3.3 Pedersen's Protocol with Proof of Possession (PedPoP) [CKM21] ... 14

3.3.4 PedPoP Reshare Protocol ... 15
4 Evaluation ... 18
4.1 ZK Proofs: Circom ... 18
References ... 19
A. Random sampling ... 20
B. Shamir Key Generation and Resharing Proposals ... 20
B.1. Proposal 1 ... 20
B.2. Proposal 2 ... 22

1 Introduction

Semaphore [Pla+24] is a zero-knowledge protocol that allows users to cast a message (e.g., a vote) as a provable member of a predetermined group, without revealing their actual identity. Internally it also produces a nullifier, which can be stored and is used to prevent users from casting a message twice (e.g., prevent double voting). The basic workflow of Semaphore is as follows: Users first create an identity (a private/public keypair) and add a so-called identity commitment to a public Merkle tree. These Merkle trees are usually managed on-chain and define the group members, as all identities committed to in the leaves of the Merkle tree are part of a group. Finally, to send a message, a user creates a zero-knowledge proof that shows: (i) they hold the secret key for a given identity, (ii) the given identity is committed to in the Merkle tree corresponding to a given public root hash and (iii) the produced nullifier is computed correctly for the given message as $H(\mathsf{sk}, \mathsf{scope})$. Since the nullifier is enforced to be computed correctly in the ZK proof, it can be used to check that a given secret key is only used once for a given scope.

The Semaphore protocol is already in version 4 and audited implementations of the circuits and client SDK exist.[1] However, for some especially long-running use-cases there exists some drawbacks as well: First, the standard Semaphore protocol equates a group member with a single identity. This lack of account abstraction makes multi-device support as well as recovery of group membership when losing a secret key difficult. Second, since the secret key of the identity is directly hashed as part of the nullifier, leakage of this secret key allows all entities to create nullifier hashes for any scopes. This obviously allows account takeover, but additionally also allows historical analysis of this accounts behavior, linking together nullifiers from different scopes.

In this document we propose a nullifier protocol that improves upon these aspects. First, as a minor change, the Merkle tree holding the accounts now has an additional layer that allows accounts to add a small number of identities in a single leaf, allowing for any of those identities to be used to create the nullifier. This introduces some problems, since now we can no longer use the secret key as part of the nullifier, since an account can now have multiple identities. To address this, we remove the secret key as part of the nullifier altogether, and use the index of the account in the Merkle tree instead. Just doing this naïvely breaks some privacy aspects of the nullifier, since now anyone could try to brute force the nullifier hash for some given index, and therefore trace actions of a specific account.

[1] See https://docs.semaphore.pse.dev for more details.

To add another secret back into the nullifier calculation, we employ an Oblivious Pseudo-random Function, where the client inputs the index into the OPRF protocol and the OPRF server holding a key k returns $F_k(i)$, without learning i. In this setup, there is now a secret k that is part of the nullifier calculation, however, it is known to the OPRF server, which could still perform the above attack. To address this, the OPRF key is secret-shared between a set of nodes, and a threshold OPRF protocol is executed instead. This protects against a malicious server, but we also need to enforce that clients cannot query arbitrary OPRF inputs, as this would allow them to calculate nullifiers of other accounts. To this end, the clients also proof in zero-knowledge that they know a secret key for a given identity in the leaf that is queried in the OPRF.

Finally, an important part of the original Semaphore protocol is the zero-knowledge proof attesting the correct calculation of the nullifier. We still require this property, but have to extend it with the correct calculation of the OPRF. Therefore, a verifiable variant of the OPRF is used, which allows the OPRF servers to prove the correct calculation of the OPRF against a known public key. This proof must then be verified in the zero-knowledge proof attesting the correct calculation of the nullifier.

1.1 Notation

In the rest of the manuscript, if not described otherwise, we refer to the prime describing the scalarfield of the BN254 curve (which is also the basefield of the BabyJubJub curve) as p and refer to the prime describing the scalarfield of the BabyJubJub curve as q. Furthermore, we denote field elements $\in \mathbb{F}$ with small letters, whereas elliptic curve points are denoted by capital letters. To denote a secret sharing of a value x or X we use a bracket notation $[x]$ and $[X]$.

2 Background

2.1 BabyJubJub

BabyJubJub is an elliptic curve designed for efficient operations inside zk-SNARKs that operate over the BN254 scalar field. The main efficiency aspects stem from the fact that the BN254 scalar field is the base field of BabyJubJub and therefore we can directly operate on the (x, y) coordinate representation in the proof system without having to use foreign field arithmetic, which is notoriously expensive. BabyJubJub is defined in EIP-2494 [WBB20], and it should be noted that there are a few conflicting definitions floating around that use slightly different, isomorphic curves instead.

> Implementation Note 1 **On the `ark-ed-on-bn254` crate** : At the time of writing, the `ark-ed-on-bn254` crate is one of these conflicting implementations, as its definition of the twisted Edwards curve is actually using the "Reduced Twisted Edwards" form from [WBB20] and is therefore incompatible, although cheap mapping functions do exist. That is why we created a new crate `ark-babyjubjub` that follows the definitions below.

The definitions below follow the EIP-2494 proposal and are compatible with existing implementations in Circom. We repeat the definitions of the BabyJubJub curve below.

2.1.1 Definiton of BabyJubJub

Let

$$p = 21888242871839275222246405745257275088548364400416034343698204186575808495617$$

and $\mathbb{F}_p$ be the finite field with p elements. p is the order of the scalar field of the elliptic curve BN254, a common pairing curve used in zk-SNARKs.

2.1.1.1 Twisted Edwards Form

Let E be the twisted Edwards elliptic curve defined over $\mathbb{F}_p$ described by the equation

$$168700x^2 + y^2 = 1 + 168696x^2y^2.$$

E is called BabyJubJub and has order

$$n = 21888242871839275222246405745257275088614511777268538073601725287587578984328,$$

which factors into $n = h \cdot q$, where the cofactor $h = 8$ and the prime

$$q = 2736030358979909402780800718157159386076813972158567259200215660948447373041.$$

The generator point G of the elliptic curve is the point of order n with

$$G = (995203441582195749578291179787384436505546430278305826713579947235728471134,\\ 5472060717959818805561601436314318772137091100104008585924551046643952123905).$$

The base point B is chosen to be $B = 8G$ and has order q. Let

$$B = (5299619240641551281634865583518297030282874472190772894086521144482721001553,\\ 16950150798460657717958625567821834550301663161624707787222815936182638968203).$$

2.1.1.2 Montgomery Form

Let E_M be the Montgomery elliptic curve defined over $\mathbb{F}_p$ described by the equation

$$v^2 = u^3 + 168698u^2 + u.$$

E_M is birationally equivalent to E, and the following mappings are used to convert points from one curve to the other.

$$E_M \mapsto E : (u,v) \to (x,y) = \left(\frac{u}{v}, \frac{u-1}{u+1}\right)$$

$$E \mapsto E_M : (x,y) \to (u,v) = \left(\frac{1+y}{1-y}, \frac{1+y}{(1-y)x}\right)$$

2.2 EdDSA on BabyJubJub

One of the main use-cases of BabyJubJub is to build a digital signature scheme from it and verify the resulting signatures in a Groth16 proof. EdDSA is somewhat standardized in RFC 8032 [JL17], however, concrete details are only given for the specific curves Curve25519 and Curve448. Furthermore, the default internal hash functions, such as SHA-512, are not zk-SNARK friendly. We instantiate an EdDSA variant using the BabyJubJub elliptic curve and the Poseidon2 [GKS23] hash function for the Fiat-Shamir transform in Algorithm 1. We also refer to [CGN20] for a rigorous treatment of EdDSA variants and follow their recommendations to achieve a strongly unforgeable variant. The variant below is also "cofactored", meaning it is amenable to batch verification.

Algorithm 1. BabyJubJub/Poseidon EdDSA signature

function KeyGen()
  $k \leftarrow \{0,1\}^{256}$ ▷ Sample random k
  $(h_0, h_1, ..., h_{511}) \leftarrow \text{Blake3}(k)$ ▷ Expand secret using hash function
  $s \leftarrow 2^{251} + h_{250} \cdot 2^{250} + ... + h_3 \cdot 2^3$ ▷ Compute secret scalar, with specific bits chosen
  $pk \leftarrow \text{sk} \cdot B$ ▷ Compute the public key
  return $\text{sk}, (h_{256}, ..., h_{511}), pk$
end

function Sign($M, \text{sk}, (h_{256}, ..., h_{511})$)
  $r \leftarrow \text{Blake3}(h_{256} \| ... \| h_{511} \| M)$ ▷ Generate a pseudorandom nonce
  $R \leftarrow r \cdot B$ ▷ Interpret r as a scalar and obtain a curve point
  $e \leftarrow \text{Poseidon2}(R \| pk \| M)$ ▷ Compute the challenge e
  $s \leftarrow r + e \cdot \text{sk} \bmod q$
  return R, s
end

function Verify($M, pk, \sigma = (R, s)$)
  Reject if $S \notin \{0, .., q-1\}$ ▷ Check for non-canonical s
  Reject if A is one of the small-order points on E.
  Reject if A or R are non-canonical.
  $e \leftarrow \text{Poseidon2}(R \| pk \| M)$ ▷ Compute the challenge e
  Accept if $8(s \cdot B - R - e \cdot pk) = 0$
end

2.3 TwoHashDH OPRF

In this paper we aim to build a distributed and verifiable OPRF service. Our main construction is derived from the TwoHashDH OPRF which was introduced in [JL10]. We give its basic construction in Scheme 1, where $H(x)$ hashes the input field element onto an elliptic curve (instantiated with BabyJubJub in our case) and $H'(\cdot)$ is a cryptographic hash function (Poseidon2 in our case).

Client(x)		Server(k)
$\beta \xleftarrow{\$} \mathbb{Z}_q$		
$A \leftarrow \beta \cdot H(x)$	$\xrightarrow{A}$	$B \leftarrow k \cdot A$
	$\xleftarrow{B}$	
Output $H'(x, (\beta^{-1} \cdot B))$		

Scheme 1. The TwoHashDH OPRF construction from [JL10].

2.4 Mapping inputs to Curve Points

The above OPRF in Scheme 1 requires a function $H : \mathbb{F}_p \mapsto E$ to encode the OPRF input into a curve point. We follow the recommendations in RFC 9380 [Faz+23] "Hashing to Elliptic Curves". For the BabyJubJub curve we are using, this internally boils down to hashing the input into a field element, calling the Elligator2 [Ber+13] mapping to get a point on the Montgomery curve E_M, mapping this point onto E via the birational map, and finally clearing the cofactor. We give an overview in Algorithm 2 and refer to more details in [Faz+23]. The `encode_to_curve` function gives a point with unknown discrete logarithm on the curve, however, the returned point is not uniform, as the Elligator2 map cannot hit about 50% of curve points. To get a truly uniform random point, the `hash_to_curve` function can be used instead, which internally maps to two points and adds them. However, for most use cases the `encode_to_curve` functions should be enough.

Algorithm 2. Hashing to elliptic curves

```
function ENCODE_TO_CURVE(x)
    u ← H(x)                              ▷ Hash the input x into a field element
    R ← elligator2_map_to_curve(u)        ▷ Use the Elligator2 map to get a point R on E_M
    Q ← birational_map(R)                 ▷ Use the birational map to get a point on E
    P ← hQ                                ▷ Clear the cofactor by multiplication.
    return P
end
```

function HASH_TO_CURVE(x)
    $u_1, u_2 \leftarrow H(x)$ ▷ Hash the input x into two field elements
    $R_1 \leftarrow$ elligator2_map_to_curve(u_1) ▷ Use the Elligator2 map to get a point R_1 on E_M
    $R_2 \leftarrow$ elligator2_map_to_curve(u_2) ▷ Use the Elligator2 map to get a point R_2 on E_M
    $R \leftarrow R_1 + R_2$
    $Q \leftarrow$ birational_map(R) ▷ Use the birational map to get a point on E
    $P \leftarrow hQ$ ▷ Clear the cofactor by multiplication.
    return P
end

2.5 Discrete Logarithm Equality Proof

Adding verifiability to Scheme 1 can be done by adding a discrete logarithm equality proof. The OPRF server is required to publish a public key $A = k \cdot D$, where D is a known point (in practice, this can just be the base point). The server then proves to the client that it has used the same key k in the OPRF evaluation [JKK14]. In the following we describe how one can prove that two group elements A and C share the same discrete logarithm k for their respective bases D and B. In other words, given $A, B, C, D \in \mathbb{G}$, we show that $A = k \cdot D$ and $C = k \cdot B$ for the same $k \in \mathbb{F}_q$. The following algorithm specifies D as an arbitrary group element, in practice D can simply be chosen as the generator of $\mathbb{G}$.

Algorithm 3. Discrete Logarithm Equality Proof

function PROVE(k, B, D)
    $r \leftarrow \mathbb{F}_q$ ▷ Sample random r
    $R_1 \leftarrow r \cdot D$
    $R_2 \leftarrow r \cdot B$
    $e \leftarrow H(k \cdot D, B, k \cdot B, D, R_1, R_2) \in \mathbb{F}_q$
    $s \leftarrow r + e \cdot k$
    return e, s
end

function VERIFY(A, B, C, D, e, s)
    Reject if $s \notin \{0, .., q-1\}$ ▷ Check for non-canonical s
    if not `validPoints`(A, B, C, D) **then**
        return $\perp$
    end
    if not `nonZeroPoints`(A, B, C, D) **then**
        return $\perp$
    end
    $R_1 \leftarrow s \cdot D - e \cdot A$
    $R_2 \leftarrow s \cdot H - e \cdot C$
    if not `nonZeroPoints`(R_1, R_2) **then**
        return $\perp$

end
$e' \leftarrow H(A, B, C, D, R_1, R_2) \in \mathbb{F}_q$
return $e = e'$
end

3 Our Construction

In this section we describe the full protocol between the client and multiple OPRF servers. We start by describing threshold variants of the OPRF construction and the discrete logarithm equality proof, before we give our full protocol with the accompanying zero-knowledge proofs.

3.1 Threshold OPRF

Translating Scheme 1 from a single-server OPRF to a threshold OPRF is trivial. Since the server (in the single server setting) only performs one group operation $B \leftarrow k \cdot A$ on a blinded A, k can just be secret-shared (e.g., using additive or Shamir [Sha79] secret-sharing) and the client reconstructs the response point B from the shares. The protocol is given in Scheme 2. Thereby, the properties of the used secret-sharing protocol (e.g., honest/dishonest majority, threshold, ect.) are inherited. For a discussion on threshold OPRFs in general we refer to [CHL22].

Client(x)		n Server($[k]$)
$\beta \xleftarrow{\$} \mathbb{Z}_q$		
$A \leftarrow \beta \cdot H(x)$	$\xrightarrow{A}$	$[B] \leftarrow [k] \cdot A$
	$\xleftarrow{[B]}$	
$B \leftarrow \texttt{Reconstruct}([B])$		
Output $H'(x, (\beta^{-1} \cdot B))$		

Scheme 2. The distributed TwoHashDH OPRF construction derived from Scheme 1.

3.2 Distributed Discrete Logarithm Equality Proof

3.2.1 Additively Shared Discrete Logarithm Equality Proof

In this section we describe how to distribute Algorithm 3 to multiple provers, where each prover has an additive secret-share of the value k. To reduce communication complexity, we introduce an accumulator party which reconstructs public values and computes challenges. Thus, each prover only has to communicate with the accumulating party, which in practice can be the verifier. We depict the protocol in Algorithm 4.

Algorithm 4. Additively Shared Discrete Logarithm Equality Proof

▷ Each server i:
function PARTIAL_COMMITMENTS(k_i, B, D)
  $r_i \leftarrow \mathbb{F}_q$ ▷ Sample random share r_i
  $R_{i,1} \leftarrow r_i \cdot D$
  $R_{i,2} \leftarrow r_i \cdot B$
  $C_i \leftarrow k_i \cdot B$
  return $r_i, C_i, R_{i,1}, R_{i,2}$
end

▷ The accumulator with input from all n servers:
function CREATE_CHALLENGE$\big(A, B, D, (C_1, R_{1,1}, R_{1,2}), ..., (C_n, R_{n,1}, R_{n,2})\big)$
  $R_1 \leftarrow R_{1,1} + ... + R_{n,1}$
  $R_2 \leftarrow R_{1,2} + ... + R_{n,2}$
  $C \leftarrow C_1 + ... + C_n$
  $e \leftarrow H(A, B, C, D, R_1, R_2) \in \mathbb{F}_q$
  return e
end

▷ Each server i on input e from the client:
function CHALLENGE(k_i, r_i, e)
  $s_i \leftarrow r_i + e \cdot k_i$
  return s_i
end

▷ The accumulator with input from all n servers:
function COMBINE_PROOFS$(e, s_1, ..., s_n)$
  $s \leftarrow s_1 + ... + s_n$
  return e, s
end

Algorithm 4 requires two communication rounds between each server and the accumulator, but the servers do not need to communicate with any other server. Thereby, each random share of the server is protected by the discrete logarithm hardness assumption, preventing the accumulator from learning anything about the secrets k, r and their shares.

In essence, Algorithm 4 rewrites the non-interactive prove back to its interactive version. Thus, the accumulator would not need to sample the random value e via the Fiat-Shamir random oracle. However, keeping the same verifier as in the non-distributed version and keeping public verifiability (i.e., proving on chain the proof was not simulated) requires the usage of the random oracle. Since the provers do not chose the challenge via Fiat-Shamir themselves, each server should only respond at most once to a challenge for an existing random share r_i.

3.2.2 Shamir Shared Discrete Logarithm Equality Proof

In order to rewrite Algorithm 4 from additive to Shamir secret-sharing, we have to make the following changes. First, the share k_i needs to be a valid Shamir share. Second, the random share r_i also needs to be sampled as a valid Shamir share, which introduces an additional communication round. Finally, the accumulator reconstructs the commitments C, R_1, R_2 and the proof s using lagrange interpolation from a set of $d+1$ servers, where d is the chosen degree of the underlying sharing polynomial. We depict the protocol in Algorithm 5.

Algorithm 5. Shamir Shared Discrete Logarithm Equality Proof

$\triangleright$ Each server i:
function PARTIAL_COMMITMENTS(k_i, B, D)
    $r_i \leftarrow$ `Shamir.Rand()` $\triangleright$ Sample random Shamir share r_i
    $R_{i,1} \leftarrow r_i \cdot D$
    $R_{i,2} \leftarrow r_i \cdot B$
    $C_i \leftarrow k_i \cdot B$
    return $r_i, C_i, R_{i,1}, R_{i,2}$
end

$\triangleright$ The accumulator with input from $t = d+1$ out of n servers:
function CREATE_CHALLENGE$\big(A, B, D, (C_1, R_{1,1}, R_{1,2}), ..., (C_t, R_{t,1}, R_{t,2})\big)$
    $R_1 \leftarrow \lambda_1 \cdot R_{1,1} + ... + \lambda_t \cdot R_{t,1}$
    $R_2 \leftarrow \lambda_1 \cdot R_{1,2} + ... + \lambda_t \cdot R_{t,2}$
    $C \leftarrow \lambda_1 \cdot C_1 + ... + \lambda_t \cdot C_t$
    $e \leftarrow H(A, B, C, D, R_1, R_2) \in \mathbb{F}_q$
    return e
end

$\triangleright$ Each server i on input e from the client:
function CHALLENGE(k_i, r_i, e)
    $s_i \leftarrow r_i + e \cdot k_i$
    return s_i
end

$\triangleright$ The accumulator with input from $t = d+1$ out of n servers:
function COMBINE_PROOFS$(e, s_1, ..., s_t)$
    $s \leftarrow \lambda_1 \cdot s_1 + ... + \lambda_t \cdot s_t$
    return e, s
end

The presence of $r_i \leftarrow$ `Shamir.Rand`() in Algorithm 5 has the implication, that the servers now need to be able to communicate with each other in order to be able to create valid Shamir shares. However, one can make a crucial optimization to get rid of this requirement again. First, observe that the only requirement on r is that it needs to be chosen to be uniform at random. Furthermore, r is only used in linear operations. Thus, when each server samples its r_i uniformly at random, each set of $d+1$ servers have a valid Shamir sharing of a random value r. Consequently, a client can just query all servers, choose the responses of $d+1$ servers to create the challenge, and use the **same** set of $d+1$ servers to construct the final proof. In this way the servers do not have to agree on all of their shares being points of the same underlying sharing polynomial and no communication between the servers is required for sampling r.

3.3 Full Distributed OPRF-Based Nullifier Protocol

We give the full verifiable OPRF based distributed nullifier service construction in Scheme 3. For the description of the zero-knowledge proofs π_1 and π_2 we refer to Section 3.3.1.

Client$(\mathrm{sk}, pk, id_u, id_{rp}, action, K)$		n Server$(k_i, K = k \cdot B)$
$\beta \xleftarrow{\$} \mathbb{Z}_q$		
$q \leftarrow H_1(id_u, id_{rp}, action)$		
$\sigma \leftarrow \mathtt{Sign}(\mathrm{sk}, q)$		
$A \leftarrow \beta \cdot H_2(q)$		
$\pi_1 \leftarrow \mathtt{prove}(\sigma, A, \mathtt{valid}(pk))$	$\xrightarrow{A, \pi_1,\ action, id_{rp}}$	if $\mathtt{verify}(\pi_1, A, action, id_{rp}) = \perp$ then abort
		$(r_i, C_i, R_{i,1}, R_{i,2}) \leftarrow$
		$\mathtt{dlog.partial\_commitments}(k_i, A, B)$
$C \leftarrow \mathtt{Reconstruct}([C])$	$\xleftarrow{[C],[R_1],[R_2]}$	
$R_1 \leftarrow \mathtt{Reconstruct}([R_1])$		
$R_2 \leftarrow \mathtt{Reconstruct}([R_2])$		
$e \leftarrow H_3(K, A, C, B, R_1, R_2) \in \mathbb{F}_p$	$\xrightarrow{e}$	$s_i \leftarrow r_i + e \cdot k_i$
	$\xleftarrow{[s]}$	
$s \leftarrow \mathtt{Reconstruct}([s])$		
$n \leftarrow H_4(q, (\beta^{-1} \cdot C))$		
$\pi_2 \leftarrow \mathtt{prove}(\sigma, A,$		
$\mathtt{valid}(pk), \mathtt{dlog.verify}(e, s), n)$		
Output (n, π_2)		

Scheme 3. The verifiable threshold TwoHashDH based nullifier construction derived from Scheme 1.

3.3.1 Clients Zero-Knowledge Proofs

We describe the ZK proofs π_1 and π_2 from Scheme 3 in this section.

3.3.1.1 Query Proof π_1

The goal of the query proof π_1 is to convince the server that a client is authorized to send a request. Therefore, $\texttt{valid}(pk)$ is a core part of this zero-knowledge proof which shows that the used public key pk is in some kind of allowlist. To prove knowledge of the corresponding private key sk we opt to verify a signature $\sigma \leftarrow \texttt{Sign}(\mathrm{sk}, q)$ of the actual query q inside the ZK proof to not have sk as a private witness in the proof. This has the advantage that a client can securely outsource proof generation to, e.g., an MPC-based prover network, without having to secret-share its key sk with them, minimizing damage in case of a privacy breach.

Finally, π_1 binds everything together by proving the correct calculation of the query point A from its parts.

In more details, proof π_1 consists of the following statements:

Proof $\boldsymbol{\pi_1} \leftarrow \texttt{prove}(\boldsymbol{\sigma}, \boldsymbol{A}, \texttt{valid}(\boldsymbol{pk}))$:

1. The query q is computed as the hash $q \leftarrow H_1(id_u, id_{rp}, action)$ where we use Poseidon2 for H_1 due to its ZK friendliness.
2. The signature σ is a valid EdDSA signature of q using some key sk. This is done by proving the EdDSA verifier inside the ZK proof, such that sk is not part of the witness.
3. The public key pk used to verify the signature σ is part of an allowlist. Concretely, this allowlist is currently implemented as a Merkle tree accumulator with the root node m and a list of t public keys at each leaf. Proving this statement is done by showing:
   - The hash of the t public keys is the actual leaf pk' of a Merkle tree
   - The prover knows a path from pk' to the root node m. This path consists of the sibling nodes in each level of the tree, as well as the position of the leaf which is id_u.
   - pk used for verifying the signature is at some index i (known to the prover) in the list of all t public keys.
4. Finally, the derivation of the query point A is computed correctly by proving $A \leftarrow \beta \cdot H_2(q)$ where $H_2(q)$ hashes q onto the BabyJubJub curve and β is a blinding element.

Fig. 2. The statements proven in π_1.

The following elements need to be public inputs to π_1:

- id_{rp} needs to be a public input, such that the OPRF servers know which secret k (which belongs to the specified RP) they need to use in their response.
- *action* is currently a public field element to bind the nullifier to a specific action publicly. If this is undesired, *action* can also be made private with no downside since *action* is part of the nullifier computation and can thus not be requested a second time.
- The Merkle root m is a public input to bind the validity check of pk to a known allowlist.
- The query point A is public such that the OPRF servers can verify the requests by the client.

3.3.1.2 Nullifier Proof π_2

The role of π_2 is having a proof of the full nullifier construction. Thus, it needs to prove the following statements:

Proof $\pi_2 \leftarrow$ `prove`$(\sigma, A,$ `valid`$(pk),$ `dlog.verify`$(e, s), n)$:

1. All 4 statements of the query proof π_1 from Fig. 2 are proven.
2. The discrete logarithm equality proof from the OPRF servers is verified to show that the servers computed the response honestly. This is done by evaluating the verifier from Algorithm 3 inside the ZK proof, similar to proving the correct EdDSA signature.
3. The OPRF result is unblinded and hashed to get the OPRF output $n = H_4(q, \beta^{-1} \cdot C)$. To avoid inverting β in the BabyJubJub scalarfield $\mathbb{F}_q$ inside a ZK proof over a different prime field $\mathbb{F}_p$ (the BabyJubJub basefield) we prove the unblinding step via injecting the result $C' = \beta^{-1} \cdot C$, showing it is a valid BabyJubJub point and proving $C = \beta \cdot C'$.
4. We allow a message $\mathsf{msg} \in \mathbb{F}_p$ to be part of the proof. To disallow tampering of the proof, we add a constraint squaring the message (as is also done in a standard Semaphore proof).

Fig. 3. The statements proven in π_2.

The following elements need to be public inputs to π_2:

- id_{rp} needs to be a public input, such that everyone can verify that the nullifier was computed for a specific RP.
- *action* is currently a public field element to bind the nullifier to a specific action publicly. If this is undesired, *action* can also be made private with no downside since *action* is part of the nullifier computation and can thus not be requested a second time.
- The OPRF public key $K = k \cdot B$ to show that the correct key k was used in the OPRF evaluation.
- The Merkle root m is a public input to bind the validity check of pk to a known allowlist.
- The message msg which is also part of the proof.
- The final nullifier n is public to bind the nullifier to the proof.

3.3.2 Key Generation and Reshare

In this section, we discuss how to generate the Shamir-shared OPRF key k and how it can be reshared with a potentially new set of key holders. This reshare protocol can also be used to refresh the existing shares by resharing to the same set of computing nodes. We begin by describing the maliciously secure distributed key generation with identifiable abort from [CKM21] (which is dubbed PedPoP) which allows to securely generate a fresh Shamir secret with any threshold of corrupted parties $t < n$.

> Remark 2 **Key generation and resharing is work in progress** : At the time of writing, it is not fully decided on how key generation and resharing will be realized and implemented. Thus, this section mainly discusses the PedPop protocol as a starting point and we refer to Section B for two proposals on how to translate this to a practical implementation. A first prototype of the final protocol will most likely be implemented using a default Shamir randomness generation, no malicious security and no resharing.

3.3.3 Pedersen's Protocol with Proof of Possession (PedPoP) [CKM21]

The PedPoP protocol is an adaption of the protocol introduced in [KG20]. It is basically a parallel instantiation of verifiable secret-sharing (VSS) based on Shamir secret-sharing with the addition of vector commitments and Schnorr proofs to ensure that communicated shares are consistent and that unforgeability holds even if more than half of the parties are corrupted. Furthermore, the protocol allows to detect and disqualify malicious parties during the key generation protocol. We depict the protocol in Fig. 4.

- **Round1**:
  1. Each party P_i chooses a random polynomial $f_i(Z)$ over $\mathbb{F}_p$ of degree t
  $$f_i(Z) = a_{i,0} + a_{i,1}Z + ... + a_{i,t}Z^t$$
  and computes $A_{i,k} = G^{a_{i,k}}$ for $k \in [t]$. Denote $x_i = a_{i,0}$ and $X_i = A_{i,0}$. Each P_i computes a proof of possession of X_i as a Schnorr signature as follows. They sample $r_i \xleftarrow{\$} \mathbb{F}_p$ and set $R_i \leftarrow G^{r_i}$. They set $c_i \leftarrow H(R_i, X_i)$ and set $z_i \leftarrow r_i + c_i \cdot x_i$. They then derive a commitment $\vec{C}_i = (A_{i,0}, ..., A_{i,t-1})$ and broadcast $\left((R_i, z_i), \vec{C}_i\right)$.
  2. After receiving the commitment from all other parties, each participant verifies the Schnorr signature by computing $c'_j \leftarrow H(R_j, A_{j,0})$ and checking that
  $$R_j A_{j,0}^{c'_j} = G^{z_j}.$$
  If any checks fail, they disqualify the corresponding participant; otherwise, they continue to the next step.
- **Round2**:
  3. Each P_i computes secret-shares $x_{i,j} = f_i(\mathrm{id}_j)$ for $j = 1, ..., n$, where id_j is the participant identifier, and sends $x_{i,j}$ secretly to party P_j.
  4. Each party P_j verifies the shares they received from the other parties by checking that
  $$G^{x_{i,j}} = \prod_{k=0}^{t} A_{i,k}^{\mathrm{id}_j^k}$$
  If the check fails for an index i, then P_j broadcasts a complaint against P_i.
- **Round3**:
  5. For each of the complaining parties P_j against P_i, P_i broadcasts the share $x_{i,j}$. If any of the revealed shares fails to satisfy the equation, or should P_i not broadcast anything for a complaining player, then P_i is disqualified. The share of a disqualified party P_i is set to 0.
- **Output**:
  6. The secret-share for each P_j is $s_j = \sum_{i=1}^{n} x_{i,j}$.
  7. If $X_i = X_j$ for any $i \neq j$, then abort. Else, the output is the joint public key $pk = \prod_{i=1}^{n} X_i$.

Fig. 4. The PedPoP key generation protocol [CKM21].

3.3.4 PedPoP Reshare Protocol

After the pair [sk], pk is generated, one can use a combined version of the reshare algorithm from [Cho+21] and the PedPoP protocol [CKM21] to reshare the key [sk] to a new set of parties, while also maintaining its correctness, privacy, and the possibility to identify malicious parties. While the original PedPoP key generation protocol requires 3 communication rounds (2 rounds for the computation plus an extra round for blaming malicious parties), our PedPoP reshare protocol only requires 2 rounds. During key generation, performing the two computation rounds in parallel would allow malicious parties to potentially introduce a bias into the random key. Since during resharing the key is already fixed, one does not need to protect against this attack vector and can perform the two communication rounds in parallel. We depict our resharing protocol in Fig. 5.

- **Round1**:
  1. To reshare the share x_i each party P_i chooses a random polynomial $f_i(Z)$ over $\mathbb{F}_p$
  $$f_i(Z) = x_i + a_{i,1}Z + ... + a_{i,t}Z^t$$
  and computes $A_{i,k} = G^{a_{i,k}}$ for $k \in [t]$. Denote $X_i = A_{i,0} = G^{x_i}$. Each P_i computes a proof of possession of X_i as a Schnorr signature as follows. They sample $r_i \xleftarrow{\$} \mathbb{F}_p$ and set $R_i \leftarrow G^{r_i}$. They set $c_i \leftarrow H(R_i, X_i)$ and set $z_i \leftarrow r_i + c_i \cdot x_i$. They then derive a commitment $\vec{C}_i = (A_{i,0}, ..., A_{i,t-1})$ and broadcast $((R_i, z_i), \vec{C}_i)$ to the new set of parties.
  2. After receiving the commitment from all old parties, each participant of the new set of parties verifies the Schnorr signature by computing $c'_j \leftarrow H(R_j, A_{j,0})$ and checking that
  $$R_j A_{j,0}^{c'_j} = G^{z_j}.$$
  Verify, that $A_{j,0}$ is equal to the commitment one receives by interpolating the commitments from Step 4 from the previous reshare round (in the exponent). If any checks fail, they accuse the corresponding participant and remove its contribution; otherwise, they continue to the next step.
- **Round2** (In parallel to Round 1):
  3. Each P_i of the old set of parties computes secret-shares $x_{i,j} = f_i(\mathrm{id}_j)$ for $j = 1, ..., n$, where id_j is the participant identifier, and sends $x_{i,j}$ secretly to party P_j of the new set.
  4. Each party P_j of the new set verifies the shares they received from the old parties:
  $$G^{x_{i,j}} = \prod_{k=0}^{t} A_{i,k}^{\mathrm{id}_j^k}$$
  If the check fails for an index i, then P_j broadcasts a complaint against P_i from the old set.
- **Round3**:
  5. For each of the complaining parties P_j against P_i, P_i broadcasts the share $x_{i,j}$. If any of the revealed shares fails to satisfy the equation, or should P_i not broadcast anything for a complaining player, then P_i is disqualified. The share of a disqualified party P_i is ignored.
- **Output**:
  6. The secret-share for each P_j is $s_j = \sum_{i=1}^{n} x_{i,j}\lambda_i$, whre λ_i is the corresponding lagrange coefficient.
  7. Check, whether the output $\prod_{i=1}^{n} X_i^{\lambda_i}$ is equal to the public key pk. This should always happen if there are at most t cheating parties.

Fig. 5. The PedPoP reshare protocol.

4 Evaluation

In this section we report some preliminary performance numbers of our proposed protocols.

4.1 ZK Proofs: Circom

We implemented the zero-knowledge proofs π_1 and π_2 from Section 3.3.1 in Circom and report on the number of R1CS constraints of parts and the full proofs in this Section. The numbers are reported in Table 1 for a Merkle tree registry of depth $d = 32$ which allows to hold $t = 7$ public keys in its leaves.

Table 1. Constraint cost for various Circom ZK building blocks.

Function	Constraint Cost	Comment
BabyJubJubScalarMulAny	2310	$s \cdot P$ for arbitrary P, 254 bit s.
BabyJubJubScalarMulFix	512	$s \cdot P$, for fixed, public P, 254 bit s.
Poseidon2 (t=2)	216	Poseidon2 with statesize 2.
Poseidon2 (t=3)	240	Poseidon2 with statesize 3.
Poseidon2 (t=4)	265	Poseidon2 with statesize 4.
Poseidon2 (t=8)	363	Poseidon2 with statesize 8.
Poseidon2 (t=16)	555	Poseidon2 with statesize 16.
EdDSAPoseidon2Verifier	5615	EdDSA Verification on BabyJubJub, using Poseidon2 as a Hash.
EncodeToCurveBabyJubJub	808	Encoding an arbitrary field element into a random BabyJubJub Curve point.
DLogEqVerify	12287	Verification of a discrete logarithm equality proof over BabyJubJub.
BinaryMerkleTree (d=32)	7875	Binary Merkle Tree using Poseidon2 with state size 2, depth 32.
Semaphore (d=32)	9383	Sempahore proof with MT depth 32.
OprfQuery (d=32)	17325	Full OPRF query proof π_1
OprfNullifier π_2 (d=32)	32414	Full OPRF nullifier proof π_2

References

[CKM21] E. C. Crites, C. Komlo, and M. Maller, "How to Prove Schnorr Assuming Schnorr: Security of Multi- and Threshold Signatures," *IACR Cryptol. ePrint Arch.*, p. 1375, 2021.

[Pla+24] V. Plasencia, A. Guzman, Ceedor, and O. Thoren, "Semaphore Protocol V4." [Online]. Available: https://github.com/zkspecs/zkspecs/blob/bdbc9b53c458bf5539069e3395e6a4e444712add/specs/3/README.md

[WBB20] B. WhiteHat, M. Bellés, and J. Baylina, "ERC-2494: Baby Jubjub Elliptic Curve." [Online]. Available: https://eips.ethereum.org/EIPS/eip-2494

[JL17] S. Josefsson and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)." [Online]. Available: https://www.rfc-editor.org/info/rfc8032

[GKS23] L. Grassi, D. Khovratovich, and M. Schofnegger, "Poseidon2: A Faster Version of the Poseidon Hash Function," in *AFRICACRYPT*, in Lecture Notes in Computer Science, vol. 14064. Springer, 2023, pp. 177–203.

[CGN20] K. Chalkias, F. Garillot, and V. Nikolaenko, "Taming the Many EdDSAs," in *SSR*, in Lecture Notes in Computer Science, vol. 12529. Springer, 2020, pp. 67–90.

[JL10] S. Jarecki and X. Liu, "Fast Secure Computation of Set Intersection," in *SCN*, in Lecture Notes in Computer Science, vol. 6280. Springer, 2010, pp. 418–435.

[Faz+23] A. Faz-Hernandez, S. Scott, N. Sullivan, R. S. Wahby, and C. A. Wood, "Hashing to Elliptic Curves." [Online]. Available: https://www.rfc-editor.org/info/rfc9380

[Ber+13] D. J. Bernstein, M. Hamburg, A. Krasnova, and T. Lange, "Elligator: elliptic-curve points indistinguishable from uniform random strings," in *CCS*, ACM, 2013, pp. 967–980.

[JKK14] S. Jarecki, A. Kiayias, and H. Krawczyk, "Round-Optimal Password-Protected Secret Sharing and T-PAKE in the Password-Only Model," in *ASIACRYPT (2)*, in Lecture Notes in Computer Science, vol. 8874. Springer, 2014, pp. 233–253.

[Sha79] A. Shamir, "How to Share a Secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[CHL22] S. Casacuberta, J. Hesse, and A. Lehmann, "SoK: Oblivious Pseudorandom Functions," in *EuroS&P*, IEEE, 2022, pp. 625–646.

[KG20] C. Komlo and I. Goldberg, "FROST: Flexible Round-Optimized Schnorr Threshold Signatures," in *SAC*, in Lecture Notes in Computer Science, vol. 12804. Springer, 2020, pp. 34–65.

[Cho+21] A. R. Choudhuri, A. Goel, M. Green, A. Jain, and G. Kaptchuk, "Fluid MPC: Secure Multiparty Computation with Dynamic Participants," in *CRYPTO (2)*, in Lecture Notes in Computer Science, vol. 12826. Springer, 2021, pp. 94–123.

Appendices

A. Random sampling

Let p be the order of the BN254 curve and q be the order of the BabyJubJub curve (see Section 2.1).

Theorem 1: Given a uniform random element $x \in \mathbb{F}_p$, the distributions $x \bmod q$ and $x \xleftarrow{\$} \mathbb{F}_q$ are statistically indistinguishable.

Proof: Let $r = 8q$, then $r \approx p + 2^{125.637}$. The distributions $x \xleftarrow{\$} \mathbb{Z}_p$ and $y \xleftarrow{\$} \mathbb{Z}_r$ are distinguishable only if the drawn element is from the gap of the two ranges, i.e. from the interval [p, r). This event happens with probability $\frac{r-p}{r} \approx \frac{2^{125.637}}{r} \approx \frac{1}{2^{127.96}}$, which is negligible, meaning our two distributions are statistically indistinguishable. For the second part of the proof, observe that r is an exact multiple of q by design. This means that the distributions $x \xleftarrow{\$} \mathbb{Z}_q;\text{return } x$ and $y \xleftarrow{\$} \mathbb{Z}_r;\text{return } y \bmod q$ are indistinguishable, since $\mathbb{Z}_q$ is a subgroup of $\mathbb{Z}_r$. Putting the two facts together concludes the proof. □

B. Shamir Key Generation and Resharing Proposals

In this section we propose two protocols on how to instantiate the Shamir secret-sharing based key generation and reshare procedures.

B.1. Proposal 1

Based on Fig. 4 we design a blockchain-assisted key generation protocol that does not require the parties to have direct communication channels with each other. However, knowledge of their respective public keys is requried. We depict this protocol in Fig. 6.

- **Round1**:
  1. Each party P_i chooses a random polynomial $f_i(Z)$ over $\mathbb{F}_p$ of degree t
  $$f_i(Z) = a_{i,0} + a_{i,1}Z + ... + a_{i,t}Z^t$$
  and computes $A_{i,k} = G^{a_{i,k}}$ for $k \in [t]$. Denote $x_i = a_{i,0}$ and $X_i = A_{i,0}$. Each P_i computes a proof of possession of X_i as a Schnorr signature as follows. They sample $r_i \xleftarrow{\$} \mathbb{F}_p$ and set $R_i \leftarrow G^{r_i}$. They set $c_i \leftarrow H(R_i, X_i)$ and set $z_i \leftarrow r_i + c_i \cdot x_i$. They then derive a commitment $\vec{C}_i = (A_{i,0}, ..., A_{i,t-1})$ and post $\left((R_i, z_i), \vec{C}_i\right)$ on chain.
  2. The smart contract verifies the Schnorr signature by computing $c'_j \leftarrow H(R_j, A_{j,0})$ and checking that
  $$R_j A_{j,0}^{c'_j} = G^{z_j}.$$
  If no error was found, the smart contract stores the commitments $\vec{C}$.
- **Round2**:
  3. Each P_i computes secret-shares $x_{i,j} = f_i(\mathrm{id}_j)$ for $j = 1, ..., n$, where id_j is the participant identifier, and posts an encryption of $x_{i,j}$ using P_j's public key on chain.
  4. Each party P_j reads the shares from the other parties from the chain and check that
  $$G^{x_{i,j}} = \prod_{k=0}^{t} A_{i,k}^{\mathrm{id}_j^k}.$$
  If the check fails for an index i, then P_j posts a complaint against P_i on chain.
- **Round3**:
  5. Each party has a predefined amount of time to post complaints on chain. For each of the complaining parties P_j against P_i, P_i broadcasts the share $x_{i,j}$. If any of the revealed shares fails to satisfy the equation, or should P_i not broadcast anything for a complaining player, then P_i is slashed and the protocol aborts.
- **Output**:
  6. The secret-share for each P_j is $s_j = \sum_{i=1}^{n} x_{i,j}$.
  7. The smart contract checks if $X_i = X_j$ for any $i \neq j$ in which case it aborts. Otherwise, it computes the public key from the commitments $pk = \prod_{i=1}^{n} X_i$.

Fig. 6. Proposal 1 for key generation based on PedPoP [CKM21].

Fig. 6 has the following (undesired) property: If party P_j files a complaint against P_i (round 3) the share $x_{i,j}$ is leaked. To prevent this, one can build the complaint round differently. If P_j makes a complaint against P_i, then P_i has to post a ZK proof that the share was derived correctly (using the commitments to the polynomials on chain) and that the encryption of the share on chain matches the correctly derived share. If that is the case, P_j filed a wrong complaint and is slashed. Otherwise, or if P_i fails to provide a proof in time, P_i is slashed.

The cost of the ZK proof is $t+1$ BabyJubJubScalarMulFix for checking the commitments, one BabyJubJubScalarMulFix plus one BabyJubJubScalarMulAny for deriving a secret key for encryption using Diffie-Hellman key agreement with the public keys of P_i and P_j, a Poseidon2 based encryption, and a recomputation of the polynomial evaluation over $\mathbb{F}_q$ for deriving the correct share. Since the polynomial evaluation is over a field which is not native to the ZK proof, this evaluation is estimated to be the most costly part of the proof. Implementing an additionModQ gadget requires an estimate of 256 constraints, whereas multiplication mod q can be implemented as a double-and-add ladder since the multiplications are with small constants (the evaluation point is the party id j) when using Horner's polynomial evaluation technique. Furthermore, this evaluation scales linearly with the degree of the polynomial t.

To allow this complaint proof to be generic for thresholds which might change later on, the circuit can be modified to an upper bound of t', and have the concrete t as a public input. This requires to Cmux each random coeefficient with 0 in case its index is greater t and recomputing t' commitments using BabyJubJubScalarMulFix. Furthermore, it could make sense to accumulate the commitments using Poseidon2 to reduce the number of public inputs.

For reshare, we modify Fig. 5 similar as for Fig. 6, with the additional constraint that the commitments to the new shares have to produce the same public key. This should come implicitly since the smart contract has to check that the correct share was used by interpolating the commitments in the exponent anyways (step 2 in Fig. 5).

Finally, for both the key generation and reshare, the proof of possession (i.e., the Schnorr proof in Step 1) is not strictly required and can be skipped.

To summarize, Proposal 1 has the advantage that it is very simple and easy to compute and does not require a direct network channel between the computing parties. Furthermore, it is publicly verifiable that the parties keep the same secret during a reshare procedure. However, the public can not verify that all parties behaved correctly without one of the parties posting a complaint on chain.

B.2. Proposal 2

The second proposal aims to achieve full verifiability with ZK proofs on chain. While the ZK proofs are more expensive compared to Proposal 1, they get rid of the complaint round and smart contracts only accept values if a ZK proof of correctness was provided. We depict the protocol in Fig. 7.

- **Round1**:
  1. Each party P_i chooses a random polynomial $f_i(Z)$ over $\mathbb{F}_p$ of degree t
  $$f_i(Z) = a_{i,0} + a_{i,1}Z + ... + a_{i,t}Z^t,$$
  computes $X_i = G^{a_{i,0}}$ and $c = H(a_{i,1}, ..., a_{i,t})$, and posts X_i and c on chain.
- **Round2**:
  2. Each P_i computes secret-shares $x_{i,j} = f_i(\mathrm{id}_j)$ for $j = 1, ..., n$, where id_j is the participant identifier, and posts a commitment $H(x_{i,j})$ and an encryption of $x_{i,j}$ using P_j's public key on chain, alongside a ZK proof verifying correctness.
  3. The smart contract verifies the ZK proof and accepts the encryptions of the shares if the proof is correct.
  4. Each party P_j reads the shares from the other parties from the chain.
- **Output**:
  6. The secret-share for each P_j is $s_j = \sum_{i=1}^{n} x_{i,j}$.
  7. The smart contract checks if $X_i = X_j$ for any $i \neq j$ in which case it aborts. Otherwise, it computes the public key from the commitments $pk = \prod_{i=1}^{n} X_i$.

Fig. 7. Proposal 2 for key generation using ZK proofs.

For reshare, we also have to proof that the correct share was used using the commitment $H(x_{i,j})$ from the previous round. When computing everything into one ZK proof (per party), the proof consists of the following:
- Deriving the n shares:
  - Requires n polynomial evaluations in a non-native field
- Recomputing the commitments:
  - Poseidon2 commitment for t inputs.
  - n Poseidon2 commitments for the shares.
  - A BabyJubJubScalarMulFix for the commitment X_i.
  - For reshare: n additional Poseidon2 commitments to verify the previous round.
- For the Encryption:
  - 1 BabyJubJubScalarMulFix for showing the secret key used for the diffie hellman matches the parties public key.
  - n BabyJubJubScalarMulAny for deriving the secret keys.
  - n Poseidon2 hashes for the encryption.

Similar to the description for Fig. 6, the cost of the ZK proof is dominated by the n polynomial evaluations in the non-native field $\mathbb{F}_q$. Furthermore, constraint size scales linearly in both the degree t and the number of parties n, making the proof very costly for larger number of parties. A first estimate for an untested implementation gives around 1.4 million R1CS constraints for $n = 30$ and $t = 15$.

Alternatively, one can think of producing n proofs for the n derived shares instead of proving everything in one large proof, which comes at the disadvantage of having to pay the on-chain gas fee n times.