A Verifiable, Threshold Oblivious Pseudorandom Function

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

Abstract.

Keywords: OPRF · VOPRF · threshold · ZKP

Contents

1 Introduction

2 Background

2.1 TwoHashDH

The TwoHashDH OPRF was introduced in [JL10] and its basic construction is given in Scheme 1.

Client(x)		Server(k)
$\beta \xleftarrow{\$} \mathbb{Z}_q$		
$a \leftarrow H(x)^\beta$	$\xrightarrow{a}$	$b \leftarrow a^k$
	$\xleftarrow{b}$	
Output $H'\left(x, b^{\beta^{-1}}\right)$		

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

2.2 BabyJubJub

BabyJubJub is an elliptic curve designed for efficient operations inside zk-SNARKs that operate over the BN254 scalar field. It 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.

Remark 1. On the `ark-ed-on-bn254` crate.

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.

2.2.1 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.

2.2.1.1 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 $n = h \cdot l$, where the cofactor $h = 8$ and the prime

$$l = 2736030358979909402780800718157159386076813972158567259200215660948447373041.$$

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 l. Let

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

2.2.1.2 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.

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

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

3 Related Work

4 Evaluation

4.1 ZK Proofs: Circom

In Table 1, we give the R1CS constraint count for various building blocks of the ZK proof.

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

Function	Constraint Cost	Comment
BabyJubJubScalarMulAny	2310	Ps for arbitrary P, 254 bit s.
BabyJubJubScalarMulFix	512	Ps, for fixed, public P, 254 bit s.
Poseidon2 (t=3)	240	Poseidon2 with statesize 3.
Poseidon2 (t=4)	264	Poseidon2 with statesize 4.
BabyJubJubPoseidonEdDSAVerify	4217	EdDSA Verification on BabyJubJub, using Poseidon as a Hash.
encode_to_curve	808	Encoding an arbitrary field element into a random BabyJubJub Curve point.
DLogEqVerify	10296	Verification of a discrete logarithm equality proof over BabyJubJub.
BinaryMerkleTree (d=32)	7875	Binary Merkle Tree using Poseidon with state size 2, depth 32.
Semaphore (d=32)	9383	Sempahore proof with MT depth 32.

4.1.1 OPRF client query validity proof

For the OPRF protocol, the client needs to proof the validity of the following statements:
1. I know a secret key sk for a given public key pk.
   - This statement is proven by providing a signature of a nonce under the given secret key sk. This signature is then verified in the ZK proof.
   - $\text{Verify}(pk, \sigma, nonce) == 1$
2. The public key pk is at the leaf with index id_u in the Merkle tree corresponding to the root hash h.

- This statement is proven by providing id_u, the Merkle tree path neighbors h_i and recomputing the path up to the root.
- $\mathrm{MTPathVerify}(pk, h_i, h)$

3. The OPrf query q is derived as specified from id_u as $H(id_u)^\beta$, for a random, but known β.
   - This statement is proven by recalculating the hash function H which hashes a field element to the BabyJubJub curve, followed by a scalar multiplication with β.
   - $\mathrm{ECScalarMul}(\mathrm{encode\_to\_curve}(id_u), \beta) = q$

The approximate circuit size for these statements is: $4217 + 7875 + 808 + 2310 = 15210 < 2^{14}$.

4.1.2 Nullifier Validity Proof

After the OPRF protocol has been executed, the client computes the nullifier $\ell = H(id_{rp,u}, action, epoch, id_{rp})$. It then proofs the validity of the whole derivation of the nullifier. This includes the steps 1-3 from above, as well as:

4. The OPRF result returned from the servers is correct w.r.t. their OPRF public key.
   - This is handled using a discrete logarithm equality proof to show that $\log_g(g^k) = \log_q(q^k)$
   - $\mathrm{DLogEqVerify}(g, g^k, q, z = q^k, s, e) == 1$
5. The OPRF result is unblinded and hashed to get the OPRF output $id_{rp,u} = H'\left(id_u, z^{\beta^{-1}}\right)$.
   - This would normally require inverting β, which is expensive since it is not native to the proof system scalar field. However, we can utilize a common trick in ZKPs and inject the result $y = z^{\beta^{-1}}$ and show that $y^\beta = z$ instead, which saves the calculation of the inverse.
   - $\mathrm{Hash2}(id_u, y) == id_{rp,u}$ and $\mathrm{ECScalarMul}(y, \beta) == z$
6. The Nullifier is calculated correctly:
   - $\mathrm{Nullifier}(id_{rp,u}, action, epoch, id_{rp})$

The approximate circuit size for these statements is: $15210 + 10296 + 240 + 2310 + 264 = 28320 < 2^{15}$.

5 Conclusion

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.

[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.

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

[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.

A. Random sampling

Let

$$p = 21888242871839275222246405745257275088548364400416034343698204186575808495617$$

be the order of the BN254 curve. Let

$$q = 2736030358979909402780800718157159386076813972158567259200215660948447373041$$

be the order of the prime-order subgroup of the BabyJubJub curve.

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 $x \xleftarrow{\$} \mathbb{Z}_q$; return x and $y \xleftarrow{\$} \mathbb{Z}_r$; 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. $\square$

B. Discrete Logarithm Equality Proof

In the following we describe how we proof that two group elements A and C share the same discrete logarithm x for their respective bases D and B. In other words, given $A, B, C, D \in \mathbb{G}$, we show that $A = x \cdot D$ and $C = x \cdot B$ for the same $x \in \mathbb{F}_p$. The following algorithm specifies D as an arbitrary group element, in practice D can simply be chosen as the generator of $\mathbb{G}$.

Algorithm 1. Prove

```
function PROVE(x, B, D)
    k ←$ F_p    ▷ Sample random k
    R_1 ← k · D
    R_2 ← k · B
```

$e \leftarrow H(x \cdot D, B, x \cdot B, D, R_1, R_2) \in \mathbb{F}_p$
$s \leftarrow k + e \cdot x$
return e, s
end

Algorithm 2. Verify

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

C. Additively Shared Discrete Logarithm Equality Proof

In this section we describe how to Distribute Algorithm 1 to multiple provers, which each have an additive secret share of the value x. 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.

Algorithm 3. Additively Shared Prove

$\triangleright$ Each server i:
function PARTIAL_COMMITMENTS(x_i, A, B, D)
$k_i \xleftarrow{\$} \mathbb{F}_p$ $\triangleright$ Sample random share k_i
$R_{i,1} \leftarrow k_i \cdot D$
$R_{i,2} \leftarrow k_i \cdot B$
$C_i \leftarrow x_i \cdot B$
return $k_i, C_i, R_{i,1}, R_{i,2}$
end

$\triangleright$ The accumulator with input from all n servers:
function CREATE_CHALLENGE$\big(A, B, D, \big(C_1, R_{\{1,1\}}, R_{\{1,2\}}\big), ..., \big(C_n, R_{\{n,1\}}, R_{\{n,2\}}\big)\big)$
$R_1 \leftarrow R_{1,1} + ... + R_{n,1}$
$R_2 \leftarrow R_{1,2} + ... + R_{n,2}$

    $C \leftarrow C_1 + ... + C_n$
    $e \leftarrow H(A, B, C, D, R_1, R_2) \in \mathbb{F}_p$
    return e
end

$\triangleright$ Each server i:
function CHALLENGE(x_i, k_i, e)
    $s_i \leftarrow k_i + e \cdot x_i$
    return s_i
end

$\triangleright$ 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 3 has 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 x, k and their shares.

In essence, Algorithm 3 rewrite 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 it is not a simulated proof) requires the usage of the random oracle. Since the prover now does not chose the challenge via Fiat-Shamir itself, each server now should only respond once to a challenge for an existing random share k_i.

D. Shamir Shared Discrete Logarithm Equality Proof

In order to rewrite Algorithm 3 from additive to Shamir secret sharing, we have to make the following changes. First, the share x_i needs to be a valid Shamir share. Second, the random share k_i also needs to be sampled as a valid Shamir shares, 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.

Algorithm 4. Shamir Shared Prove

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

return $k_i, C_i, R_{i,1}, R_{i,2}$
end

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

▷ Each server i:
function CHALLENGE(x_i, k_i, e)
$s_i \leftarrow k_i + e \cdot x_i$
return s_i
end

▷ 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 $k_i \leftarrow +$ `Shamir.Rand()` in Algorithm 4 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. In practices one can think of either generating this random share directly on request, or precomputing random values k in an offline phase and consuming them in the online phase. Another solution can be to let the accumulator chose the $d + 1$ servers in the beginning of the protocol and only use their shares in the whole computation. In this setting, the chosen parties can simply sample their shares at random without communication, since the requirement that all n shares need to be on the same polynomial is not there anymore.

D.1. Shamir Key Generation and Resharing

In this section we describe how to generate the Shamir-shared OPRF key and how to reshare it with a potentially new set of key holders. The reshare protocol can also be used to refresh the shares of the OPRF key by resharing to the same set of computing nodes. We begin by describing the maliciously secure distributed key generation with identifiable abort which allows to securely generate a fresh Shamir secret with any threshold of corrupted parties $t < n$.

D.1.1. 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. 2.

- **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 $\vec{C}_i = \left(A_{i,0}, ..., A_{i,t-1}\right)$ 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 $c'_j \leftarrow H\left(R_j, A_{j,0}\right)$ 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 $x_{i,j} = f_i\left(\mathrm{id}_j\right)$ 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. 2. The PedPoP key generation protocol [CKM21].

D.1.2. 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. 3.

- **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. 3. The PedPoP reshare protocol.

D.1.3. Proposal 1

Based on Fig. 2 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 public keys is requried.

- **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 posts $\left((R_i, z_i), \vec{C}_i\right)$ 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 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 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. 4. Proposal 1 for key generation based on PedPoP [CKM21].

Fig. 4 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, two BabyJubJubScalarMulAny for deriving a secret key for encryption, and a Poseidon based encryption. Thus, the cost of this ZK proof is approximately $5500 + t \cdot 512$ constraints, which is less than 2^{14} for $t = 15$. To allow this complained 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 coeeficient 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. 3 similar as for Fig. 4, 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. 3).

Furthermore, 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.

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.

D.1.4. 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.

- **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. 5. 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:

- 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:
  - $2 \cdot n$ BabyJubJubScalarMulAny for deriving the secret keys
  - n Poseidon2 hashes for the encryption
- In total:
  - $3 \cdot n + t$ Poseidon2 with 240 constraints each
  - 1 BabyJubJubScalarMulFix with 512 constraints
  - $2 \cdot n$ BabyJubJubScalarMulAny with 2310 constraints

This totals to approximately 165000 ($< 2^{18}$) constraints for $n = 30$ with $t = 15$, where the majority with 140k constraints comes from the BabyJubJubScalarMulAny gadget.

Alternatively, one can think of producing n proofs for the n derived shares instead of proving everything in one large proof.