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].

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.
   - $\mathsf{Verify}(pk, \sigma, nonce) == 1$
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.
   - $\mathsf{MTPathVerify}(pk, h_i, h)$
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.
   - $\mathsf{ECScalarMul}(\mathsf{encode\_to\_curve}(id_u), \beta) = q$

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 $\ell = H\left(id_{rp,u}, action, epoch, id_{rp}\right)$. 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 $\log_g\left(g^k\right) = \log_q\left(q^k\right)$
   - $\text{DLogEqVerify}\left(g, g^k, q, z = q^k, s, e\right) == 1$
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.
   - $\text{Hash2}(id_u, y) == id_{rp,u}$ and $\text{ECScalarMul}(y, \beta) == z$
6. The Nullifier is calculated correctly:
   - $\text{Nullifier}\left(id_{rp,u}, action, epoch, id_{rp}\right)$

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

5 Conclusion

References

[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.

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 $x \xleftarrow{\$} \mathbb{Z}_q; \text{return } x$ 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. 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($x, B, D$)
    $k \xleftarrow{\$} \mathbb{F}_p$  ▷ Sample random $k$
    $R_1 \leftarrow k \cdot D$
    $R_2 \leftarrow k \cdot B$
    $e \leftarrow H(x \cdot D, B, x \cdot B, D, R_1, R_2) \in \mathbb{F}_p$
    $s \leftarrow k + e \cdot x$
    return $e, s$
end
```

Algorithm 2. Verify

```
function Verify($A, B, C, D, e, s$)
    if not validPoints($A, B, C, D$) then
        return $\bot$
    end
    if not nonZeroPoints($A, B, C, D$) then
        return $\bot$
    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 $\bot$
    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

```
▷ Each server $i$:
function PARTIAL_COMMITMENTS($x_i, A, B, D$)
    $k_i \xleftarrow{\$} \mathbb{F}_p$   ▷ 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

▷ 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

▷ 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 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 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

```
▷ Each server $i$:
function PARTIAL_COMMITMENTS($x_i, A, B, D$)
    $k_i \leftarrow$ Shamir.Rand()    ▷ 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.