Network Working Group
Internet-Draft
Intended status: Informational
Expires: January 9, 2020
A. Davidson
N. Sullivan
Cloudflare
C. Wood
Apple Inc.
July 08, 2019

Oblivious Pseudorandom Functions (OPRFs) using Prime-Order Groups
draft-irtf-cfrg-voprf-00

Abstract

An Oblivious Pseudorandom Function (OPRF) is a two-party protocol for computing the output of a PRF. One party (the server) holds the PRF secret key, and the other (the client) holds the PRF input. The 'obliviousness' property ensures that the server does not learn anything about the client's input during the evaluation. The client should also not learn anything about the server's secret PRF key. Optionally, OPRFs can also satisfy a notion 'verifiability' (VOPRF). In this setting, the client can verify that the server's output is indeed the result of evaluating the underlying PRF with just a public key. This document specifies OPRF and VOPRF constructions instantiated within prime-order groups, including elliptic curves.

Table of Contents

1. Introduction

A pseudorandom function (PRF) F(k, x) is an efficiently computable function with secret key k on input x. Roughly, F is pseudorandom if the output y = F(k, x) is indistinguishable from uniformly sampling any element in F's range for random choice of k. An oblivious PRF (OPRF) is a two-party protocol between a prover P and verifier V where P holds a PRF key k and V holds some input x. The protocol allows both parties to cooperate in computing F(k, x) with P's secret key k and V's input x such that: V learns F(k, x) without learning anything about k; and P does not learn anything about x. A Verifiable OPRF (VOPRF) is an OPRF wherein P can prove to V that F(k, x) was computed using key k, which is bound to a trusted public key Y = kG. Informally, this is done by presenting a non-interactive zero-knowledge (NIZK) proof of equality between (G, Y) and (Z, M), where Z = kM for some point M.

OPRFs have been shown to be useful for constructing: password-protected secret sharing schemes [JKK14]; privacy-preserving password stores [SJKS17]; and password-authenticated key exchange or PAKE [OPAQUE]. VOPRFs are useful for producing tokens that are verifiable by V. This may be needed, for example, if V wants assurance that P did not use a unique key in its computation, i.e., if V wants key consistency from P. This property is necessary in some applications, e.g., the Privacy Pass protocol [PrivacyPass], wherein this VOPRF is used to generate one-time authentication tokens to bypass CAPTCHA challenges. VOPRFs have also been used for password-protected secret sharing schemes e.g. [JKKX16].

This document introduces an OPRF protocol built in prime-order groups, applying to finite fields of prime-order and also elliptic curve (EC) settings. The protocol has the option of being extended

to a VOPRF with the addition of a NIZK proof for proving discrete log equality relations. This proof demonstrates correctness of the computation using a known public key that serves as a commitment to the server's secret key. In the EC setting, we will refer to the protocol as ECOPRF (or ECVOPRF if verifiability is concerned). The document describes the protocol, its security properties, and provides preliminary test vectors for experimentation. The rest of the document is structured as follows:

- Section 2: Describe background, related work, and use cases of OPRF/VOPRF protocols.

- Section 3: Discuss security properties of OPRFs/VOPRFs.

- Section 4: Specify an authentication protocol from OPRF functionality, based in prime-order groups (with an optional verifiable mode). Algorithms are stated formally for OPRFs in Section 4.3 and for VOPRFs in Section 4.4.

- Section 5: Specify the NIZK discrete logarithm equality (DLEQ) construction used for constructing the VOPRF protocol.

- Section 6: Specifies how the DLEQ proof mechanism can be batched for multiple VOPRF invocations, and how this changes the protocol execution.

- Section 7: Considers explicit instantiations of the protocol in the elliptic curve setting.

- Section 8: Discusses the security considerations for the OPRF and VOPRF protocol.

- Section 9: Discusses some existing applications of OPRF and VOPRF protocols.

- Appendix A: Specifies test vectors for implementations in the elliptic curve setting.

1.1. Terminology

The following terms are used throughout this document.

- PRF: Pseudorandom Function.

- OPRF: Oblivious PRF.

- VOPRF: Verifiable Oblivious Pseudorandom Function.

- o ECVOPRF: A VOPRF built on Elliptic Curves.

- o Verifier (V): Protocol initiator when computing F(k, x).

- o Prover (P): Holder of secret key k.

- o NIZK: Non-interactive zero knowledge.

- o DLEQ: Discrete Logarithm Equality.

1.2. Requirements

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. Background

OPRFs are functionally related to blind signature schemes. In such a scheme, a client can receive signatures on private data, under the signing key of some server. The security properties of such a scheme dictate that the client learns nothing about the signing key, and that the server learns nothing about the data that is signed. One of the more popular blind signature schemes is based on the RSA cryptosystem and is known as Blind RSA [ChaumBlindSignature].

OPRF protocols can thought of as symmetric alternatives to blind signatures. Essentially the client learns y = PRF(k,x) for some input x of their choice, from a server that holds k. Since the security of an OPRF means that x is hidden in the interaction, then the client can later reveal x to the server along with y.

The server can verify that y is computed correctly by recomputing the PRF on x using k. In doing so, the client provides knowledge of a 'signature' y for their value x. The verification procedure is thus symmetric as it requires knowledge of the key k. This is discussed more in the following section.

3. Security Properties

The security properties of an OPRF protocol with functionality y = F(k, x) include those of a standard PRF. Specifically:

- o Pseudorandomness: F is pseudorandom if the output y = F(k,x) on any input x is indistinguishable from uniformly sampling any element in F's range, for a random sampling of k.

In other words, for an adversary that can pick inputs x from the domain of F and can evaluate F on (k,x) (without knowledge of randomly sampled k), then the output distribution F(k,x) is indistinguishable from the uniform distribution in the range of F.

A consequence of showing that a function is pseudorandom, is that it is necessarily non-malleable (i.e. we cannot compute a new evaluation of F from an existing evaluation). A genuinely random function will be non-malleable with high probability, and so a pseudorandom function must be non-malleable to maintain indistinguishability.

An OPRF protocol must also satisfy the following property:

- Oblivious: P must learn nothing about V's input or the output of the function. In addition, V must learn nothing about P's private key.

Essentially, obliviousness tells us that, even if P learns V's input x at some point in the future, then P will not be able to link any particular OPRF evaluation to x. This property is also known as unlinkability [DGSTV18].

Optionally, for any protocol that satisfies the above properties, there is an additional security property:

- Verifiable: V must only complete execution of the protocol if it can successfully assert that the OPRF output computed by V is correct, with respect to the OPRF key held by P.

Any OPRF that satisfies the 'verifiable' security property is known as a verifiable OPRF, or VOPRF for short. In practice, the notion of verifiability requires that P commits to the key k before the actual protocol execution takes place. Then V verifies that P has used k in the protocol using this commitment. In the following, we may also refer to this commitment as a public key.

4. OPRF Protocol

In this section we describe the OPRF protocol. Let GG be an additive group of prime-order p, let GF(p) be the Galois field defined by the integers modulo p. Define distinct hash functions H_1 and H_2, where H_1 maps arbitrary input onto GG and H_2 maps arbitrary input to a fixed-length output, e.g., SHA256. All hash functions in the protocol are modelled as random oracles. Let L be the security parameter. Let k be the prover's (P) secret key, and Y = kG be its corresponding 'public key' for some fixed generator G taken from the description of the group GG. This public key Y is also referred to as a commitment to the OPRF key k, and the pair (G,Y) as a commitment

pair. Let x be the verifier's (V) input to the OPRF protocol. (Commonly, it is a random L-bit string, though this is not required.)

The OPRF protocol begins with V blinding its input for the OPRF evaluator such that it appears uniformly distributed GG. The latter then applies its secret key to the blinded value and returns the result. To finish the computation, V then removes its blind and hashes the result using H_2 to yield an output. This flow is illustrated below.

```
     Verifier                              Prover
  ----------------------------------------------------------
     r <-$ GF(p)
     M = rH_1(x) mod p
                          M
                        ------->
                                 Z = kM mod p
                                 [D = DLEQ_Generate(k,G,Y,M,Z)]
                          Z[,D]
                        <-------
    [b = DLEQ_Verify(G,Y,M,Z,D)]
    N = Zr^(-1) mod p
    Output H_2(x, N) mod p [if b=1, else "error"]
```

Steps that are enclosed in square brackets (DLEQ_Generate and DLEQ_Verify) are optional for achieving verifiability. These are described in Section 5. In the verifiable mode, we assume that P has previously committed to their choice of key k with some values (G,Y=kG) and these are publicly known by V. Notice that revealing (G,Y) does not reveal k by the well-known hardness of the discrete log problem.

Strictly speaking, the actual PRF function that is computed is:

F(k, x) = N = kH_1(x)

It is clear that this is a PRF H_1(x) maps x to a random element in GG, and GG is cyclic. This output is computed when the client computes Zr^(-1) by the commutativity of the multiplication. The client finishes the computation by outputting H_2(x,N). Note that the output from P is not the PRF value because the actual input x is blinded by r.

This protocol may be decomposed into a series of steps, as described below:

o OPRF_Setup(l): Generate am integer k of sufficient bit-length l and output k.

o OPRF_Blind(x): Compute and return a blind, r, and blinded representation of x in GG, denoted M.

o OPRF_Eval(k,M,h?): Evaluates on input M using secret key k to produce Z, the input h is optional and equal to the cofactor of an elliptic curve. If h is not provided then it defaults to 1.

o OPRF_Unblind(r,Z): Unblind blinded OPRF evaluation Z with blind r, yielding N and output N.

o OPRF_Finalize(x,N): Finalize N to produce the output H_2(x, N).

For verifiability we modify the algorithms of VOPRF_Setup, VOPRF_Eval and VOPRF_Unblind to be the following:

o VOPRF_Setup(l): Generate an integer k of sufficient bit-length l and output (k, (G,Y)) where Y = kG for the fixed generator G of GG.

o VOPRF_Eval(k,(G,Y),M,h?): Evaluates on input M using secret key k to produce Z. Generate a NIZK proof D = DLEQ_Generate(k,G,Y,M,Z), and output (Z, D). The optional cofactor h can also be provided, as in OPRF_Eval.

o VOPRF_Unblind(r,G,Y,M,(Z,D)): Unblind blinded OPRF evaluation Z with blind r, yielding N. Output N if 1 = DLEQ_Verify(G,Y,M,Z,D). Otherwise, output "error".

We leave the rest of the OPRF algorithms unmodified. When referring explicitly to VOPRF execution, we replace 'OPRF' in all method names with 'VOPRF'.

4.1. Protocol correctness

Protocol correctness requires that, for any key k, input x, and (r, M) = OPRF_Blind(x), it must be true that:

```
OPRF_Finalize(x, OPRF_Unblind(r,M,OPRF_Eval(k,M))) = H_2(x, F(k,x))
```

with overwhelming probability. Likewise, in the verifiable setting, we require that:

```
VOPRF_Finalize(x, VOPRF_Unblind(r,(G,Y),M,(VOPRF_Eval(k,(G,Y),M)))) = H_2(x,
F(k,x))
```

with overwhelming probability, where (r, M) = VOPRF_Blind(x).

4.2. Instantiations of GG

As we remarked above, GG is a subgroup with associated prime-order p. While we choose to write operations in the setting where GG comes equipped with an additive operation, we could also define the operations in the multiplicative setting. In the multiplicative setting we can choose GG to be a prime-order subgroup of a finite field FF_p. For example, let p be some large prime (e.g. > 2048 bits) where p = 2q+1 for some other prime q. Then the subgroup of squares of FF_p (elements u^2 where u is an element of FF_p) is cyclic, and we can pick a generator of this subgroup by picking G from FF_p (ignoring the identity element).

For practicality of the protocol, it is preferable to focus on the cases where GG is an additive subgroup so that we can instantiate the OPRF in the elliptic curve setting. This amounts to choosing GG to be a prime-order subgroup of an elliptic curve over base field GF(p) for prime p. There are also other settings where GG is a prime-order subgroup of an elliptic curve over a base field of non-prime order, these include the work of Ristretto [RISTRETTO] and Decaf [DECAF].

We will use p > 0 generally for constructing the base field GF(p), not just those where p is prime. To reiterate, we focus only on the additive case, and so we focus only on the cases where GF(p) is indeed the base field.

Unless otherwise stated, we will always assume that the generator G that we use for the group GG is a fixed generator. This generator should be provided in the description of the group GG.

4.3. OPRF algorithms

This section provides algorithms for each step in the OPRF protocol. We describe the VOPRF analogues in Section 4.4. We provide generic utility algorithms in Section 4.5.

1. P samples a uniformly random key k <- {0,1}^l for sufficient length l, and interprets it as an integer.

2. V computes X = H_1(x) and a random element r (blinding factor) from GF(p), and computes M = rX.

3. V sends M to P.

4. P computes Z = kM = rkX.

5. In the elliptic curve setting, P multiplies Z by the cofactor (denoted h) of the elliptic curve.

6. P sends Z to V.

7. V unblinds Z to compute N = r^(-1)Z = kX.

8. V outputs the pair H_2(x, N).

We note here that the blinding mechanism that we use can be modified slightly with the opportunity for making performance gains in some scenarios. We detail these modifications in Section Section 4.6.

4.3.1. OPRF_Setup

Input:

```
 l: Some suitable choice of key-length (e.g. as described in [NIST]).
```

Output:

```
 k: A key chosen from {0,1}^l and interpreted as an integer value.
```

Steps:

```
 1. Sample k_bin <-$ {0,1}^l
 2. Output k <- bin2scalar(k_bin, l)
```

4.3.2. OPRF_Blind

Input:

```
 x: V's PRF input.
```

Output:

```
 r: Random scalar in [1, p - 1].
 M: Blinded representation of x using blind r, an element in GG.
```

Steps:

```
 1.  r <-$ GF(p)
 2.  M := rH_1(x)
 3.  Output (r, M)
```

4.3.3. OPRF_Eval

```
Input:

 k: Evaluator secret key.
 M: An element in GG.
 h: optional cofactor (defaults to 1).

Output:

 Z: Scalar multiplication of the point M by k, element in GG.

Steps:

 1. Z := kM
 2. Z <- hZ
 3. Output Z
```

4.3.4. OPRF_Unblind

```
Input:

 r: Random scalar in [1, p - 1].
 Z: An element in GG.

Output:

 N: Unblinded OPRF evaluation, element in GG.

Steps:

 1. N := (r^(-1))Z
 2. Output N
```

4.3.5. OPRF_Finalize

```
Input:

 x: PRF input string.
 N: An element in GG.

Output:

 y: Random element in {0,1}^L.

Steps:

 1. y := H_2(x, N)
 2. Output y
```

4.4. VOPRF algorithms

The steps in the VOPRF setting are written as:

1. P samples a uniformly random key k <- {0,1}^l for sufficient length l, and interprets it as an integer.

2. P commits to k by computing (G,Y) for Y=kG, where G is the fixed generator of GG. P makes the pair (G,Y) publicly available.

3. V computes X = H_1(x) and a random element r (blinding factor) from GF(p), and computes M = rX.

4. V sends M to P.

5. P computes Z = kM = rkX, and D = DLEQ_Generate(k,G,Y,M,Z).

6. P sends (Z, D) to V.

7. V ensures that 1 = DLEQ_Verify(G,Y,M,Z,D). If not, V outputs an error.

8. V unblinds Z to compute N = r^(-1)Z = kX.

9. V outputs the pair H_2(x, N).

4.4.1. VOPRF_Setup

Input:

```
 G: Public fixed generator of GG.
 l: Some suitable choice of key-length (e.g. as described in [NIST]).
```

Output:

```
 k: A key chosen from {0,1}^l and interpreted as an integer value.
 (G,Y): A pair of curve points, where Y=kG.
```

Steps:

```
  1. k <- OPRF_Setup(l)
  2. Y := kG
  3. Output (k, (G,Y))
```

4.4.2. VOPRF_Blind

Input:

 x: V's PRF input.

Output:

 r: Random scalar in [1, p - 1].
 M: Blinded representation of x using blind r, an element in GG.

Steps:

```
1.  r <-$ GF(p)
2.  M := rH_1(x)
3.  Output (r, M)
```

4.4.3. VOPRF_Eval

Input:

 k: Evaluator secret key.
 G: Public fixed generator of group GG.
 Y: Evaluator public key (= kG).
 M: An element in GG.
 h: optional cofactor (defaults to 1).

Output:

 Z: Scalar multiplication of the point M by k, element in GG.
 D: DLEQ proof that log_G(Y) == log_M(Z).

Steps:

```
1. Z := kM
2. Z <- hZ
3. D = DLEQ_Generate(k,G,Y,M,Z)
4. Output (Z, D)
```

4.4.4. VOPRF_Unblind

Input:

r: Random scalar in [1, p - 1].
G: Public fixed generator of group GG.
Y: Evaluator public key.
M: Blinded representation of x using blind r, an element in GG.
Z: An element in GG.
D: D = DLEQ_Generate(k,G,Y,M,Z).

Output:

N: Unblinded OPRF evaluation, element in GG.

Steps:

1. N := (r^(-1))Z
2. If 1 = DLEQ_Verify(G,Y,M,Z,D), output N
3. Output "error"

4.4.5. VOPRF_Finalize

Input:

x: PRF input string.
N: An element in GG, or "error".

Output:

y: Random element in {0,1}^L, or "error"

Steps:

1. If N == "error", output "error".
2. y := H_2(x, N)
3. Output y

4.5. Utility algorithms

4.5.1. bin2scalar

This algorithm converts a binary string to an integer modulo p.

Input:

```
 s: binary string (little-endian)
 l: length of binary string
 p: modulus
```

Output:

```
 z: An integer modulo p
```

Steps:

```
 1. sVec <- vec(s) (converts s to a column vector of dimension l)
 2. p2Vec <- (2^0, 2^1, ..., 2^{l-1}) (row vector of dimension l)
 3. z <- p2Vec * sVec (mod p)
 4. Output z
```

4.6. Efficiency gains with pre-processing and fixed-base blinding

In Section Section 4.3 we assume that the client-side blinding is carried out directly on the output of H_1(x), i.e. computing rH_1(x) for some r <-$ GF(p). In the [OPAQUE] draft, it is noted that it may be more efficient to use additive blinding rather than multiplicative if the client can preprocess some values. For example, a valid way of computing additive blinding would be to instead compute H_1(x)+rG, where G is the fixed generator for the group GG.

We refer to the 'multiplicative' blinding as variable-base blinding (VBB), since the base of the blinding (H_1(x)) varies with each instantiation. We refer to the additive blinding case as fixed-base blinding (FBB) since the blinding is applied to the same generator each time (when computing rG).

By pre-processing tables of blinded scalar multiplications for the specific choice of G it is possible to gain a computational advantage. Choosing one of these values rG (where r is the scalar value that is used), then computing H_1(x)+rG is more efficient than computing rH_1(x) (one addition against log_2(r)). Therefore, it may be advantageous to define the OPRF and VOPRF protocols using additive blinding rather than multiplicative blinding. In fact, the only algorithms that need to change are OPRF_Blind and OPRF_Unblind (and similarly for the VOPRF variants).

We define the FBB variants of the algorithms in Section 4.3 below along with a new algorithm OPRF_Preprocess that defines how preprocessing is carried out. The equivalent algorithms for VOPRF are almost identical and so we do not redefine them here. Notice

that the only computation that changes is for V, the necessary computation of P does not change.

4.6.1. OPRF_Preprocess

Input:

```
 G: Public fixed generator of GG
```

Output:

```
 r: Random scalar in [1, p-1]
 rG: An element in GG.
 rY: An element in GG.
```

Steps:

```
 1.  r <-$ GF(p)
 2.  Output (r, rG, rY)
```

4.6.2. OPRF_Blind

Input:

```
 x: V's PRF input.
 rG: Preprocessed element of GG.
```

Output:

```
 M: Blinded representation of x using blind r, an element in GG.
```

Steps:

```
 1.  M := H_1(x)+rG
 2.  Output M
```

4.6.3. OPRF_Unblind

Input:

rY: Preprocessed element of GG.
M: Blinded representation of x using rG, an element in GG.
Z: An element in GG.

Output:

N: Unblinded OPRF evaluation, element in GG.

Steps:

1. N := Z-rY
2. Output N

Notice that OPRF_Unblind computes (Z-rY) = k(H_1(x)+rG) - rkG = kH_1(x) by the commutativity of scalar multiplication in GG. This is the same output as in the original OPRF_Unblind algorithm.

5. NIZK Discrete Logarithm Equality Proof

For the VOPRF protocol we require that V is able to verify that P has used its private key k to evaluate the PRF. We can do this by showing that the original commitment (G,Y) output by VOPRF_Setup(l) satisfies log_G(Y) == log_M(Z) where Z is the output of VOPRF_Eval(k,(G,Y),M).

This may be used, for example, to ensure that P uses the same private key for computing the VOPRF output and does not attempt to "tag" individual verifiers with select keys. This proof must not reveal the P's long-term private key to V.

Consequently, this allows extending the OPRF protocol with a (non-interactive) discrete logarithm equality (DLEQ) algorithm built on a Chaum-Pedersen [ChaumPedersen] proof. This proof is divided into two procedures: DLEQ_Generate and DLEQ_Verify. These are specified below.

5.1. DLEQ_Generate

Input:

k: Evaluator secret key.
G: Public fixed generator of GG.
Y: Evaluator public key (= kG).
M: An element in GG.
Z: An element in GG.
H_3: A hash function from GG to {0,1}^L, modelled as a random oracle.

Output:

D: DLEQ proof (c, s).

Steps:

1. r <-$ GF(p)
2. A := rG and B := rM.
3. c <- H_3(G,Y,M,Z,A,B)
4. s := (r - ck) (mod p)
5. Output D := (c, s)

We note here that it is essential that a different r value is used for every invocation. If this is not done, then this may leak the key k in a similar fashion as is possible in Schnorr or (EC)DSA scenarios where fresh randomness is not used.

5.2. DLEQ_Verify

Input:

G: Public fixed generator of GG.
Y: Evaluator public key.
M: An element in GG.
Z: An element in GG.
D: DLEQ proof (c, s).

Output:

True if log_G(Y) == log_M(Z), False otherwise.

Steps:

1. A' := (sG + cY)
2. B' := (sM + cZ)
3. c' <- H_3(G,Y,M,Z,A',B')
4. Output c == c'

6. Batched VOPRF evaluation

Common applications (e.g. [PrivacyPass]) require V to obtain multiple PRF evaluations from P. In the VOPRF case, this would also require generation and verification of a DLEQ proof for each Zi received by V. This is costly, both in terms of computation and communication. To get around this, applications use a 'batching' procedure for generating and verifying DLEQ proofs for a finite number of PRF evaluation pairs (Mi,Zi). For n PRF evaluations:

- Proof generation is slightly more expensive from 2n modular exponentiations to 2n+2.

- Proof verification is much more efficient, from 4n modular exponentiations to 2n+4.

- Communications falls from 2n to 2 group elements.

Therefore, since P is usually a powerful server, we can tolerate a slight increase in proof generation complexity for much more efficient communication and proof verification.

In this section, we describe algorithms for batching the DLEQ generation and verification procedure. For these algorithms we require an additional random oracle H_5: {0,1}^a x ZZ^3 -> {0,1}^b that takes an inputs of a binary string of length a and three integer values, and outputs an element in {0,1}^b.

6.1. Batched DLEQ algorithms

6.1.1. Batched_DLEQ_Generate

```
Input:

 k: Evaluator secret key.
 G: Public fixed generator of group GG.
 Y: Evaluator public key (= kG).
 n: Number of PRF evaluations.
 [ Mi ]: An array of points in GG of length n.
 [ Zi ]: An array of points in GG of length n.
 H_4: A hash function from GG^(2n+2) to {0,1}^a, modelled as a random oracle.
 H_5: A hash function from {0,1}^a x ZZ^2 to {0,1}^b, modelled as a random
oracle.
 label: An integer label value for the splitting the domain of H_5

Output:

 D: DLEQ proof (c, s).

Steps:

 1. seed <- H_4(G,Y,[Mi,Zi]))
 2. for i in [n]: di <- H_5(seed,i,label)
 3. c1,...,cn := (int)d1,...,(int)dn
 4. M := c1M1 + ... + cnMn
 5. Z := c1Z1 + ... + cnZn
 6. Output D <- DLEQ_Generate(k,G,Y,M,Z)
```

6.1.2. Batched_DLEQ_Verify

```
 Input:

  G: Public fixed generator of group GG.
  Y: Evaluator public key.
  [ Mi ]: An array of points in GG of length n.
  [ Zi ]: An array of points in GG of length n.
  D: DLEQ proof (c, s).

 Output:

  True if log_G(Y) == log_(Mi)(Zi) for each i in 1...n, False otherwise.

 Steps:

  1. seed <- H_4(G,Y,[Mi,Zi]))
  2. for i in [n]: di <- H_5(seed,i,info)
  3. c1,...,cn := (int)d1,...,(int)dn
  4. M := c1M1 + ... + cnMn
  5. Z := c1Z1 + ... + cnZn
  6. Output DLEQ_Verify(G,Y,M,Z,D)
```

6.2. Modified protocol execution

The VOPRF protocol from Section Section 4 changes to allow specifying multiple blinded PRF inputs [Mi] for i in 1...n. P computes the array [Zi] and replaces DLEQ_Generate with Batched_DLEQ_Generate over these arrays. The same applies to the algorithm VOPRF_Eval. The same applies for replacing DLEQ_Verify with Batched_DLEQ_Verify when V verifies the response from P and during the algorithm VOPRF_Verify.

6.3. Random oracle instantiations for proofs

We can instantiate the random oracle function H_4 using the same hash function that is used for H_1,H_2,H_3. For H_5, we can also use a similar instantiation, or we can use a variable-length output generator. For example, for groups with an order of 256-bit, valid instantiations include functions such as SHAKE-256 [SHAKE] or HKDF-Expand-SHA256 [RFC5869].

In addition if a function with larger output than the order of the base field is used, we note that the outputs of H_5 (d1,...,dn) must be smaller than this order. If any di that is sampled is larger than then order, then we should resample until a di' is sampled that is valid.

In these cases, the iterating integer i is increased monotonically to i' until such di' is sampled. When sampling the next value d(i+1), the counter i+1 is started at i'+1.

TODO: Give a more detailed specification of this construction.

7. Supported ciphersuites

This section specifies supported ECVOPRF group and hash function instantiations. We only provide ciphersuites in the EC setting as these provide the most efficient way of instantiating the OPRF. Our instantiation includes considerations for providing the DLEQ proofs that make the instantiation a VOPRF. Supporting OPRF operations (ECOPRF) alone can be allowed by simply dropping the relevant components. In addition, we currently only support ciphersuites demonstrating 128 bits of security.

7.1. ECVOPRF-P256-HKDF-SHA256-SSWU:

- GG: SECP256K1 curve [SEC2]

- H_1: H2C-P256-SHA256-SSWU- [I-D.irtf-cfrg-hash-to-curve]

      * label: voprf_h2c

   o H_2: SHA256

   o H_3: SHA256

   o H_4: SHA256

   o H_5: HKDF-Expand-SHA256

7.2. ECVOPRF-RISTRETTO-HKDF-SHA512-Elligator2:

   o GG: Ristretto [RISTRETTO]

   o H_1: H2C-Curve25519-SHA512-Elligator2-Clear [I-D.irtf-cfrg-hash-to-curve]

      * label: voprf_h2c

   o H_2: SHA512

   o H_3: SHA512

   o H_4: SHA512

   o H_5: HKDF-Expand-SHA512

In the case of Ristretto, internal point representations are represented by Ed25519 [RFC7748] points. As a result, we can use the same hash-to-curve encoding as we would use for Ed25519 [I-D.irtf-cfrg-hash-to-curve]. We remark that the 'label' field is necessary for domain separation of the hash-to-curve functionality.

8. Security Considerations

Security of the protocol depends on P's secrecy of k. Best practices recommend P regularly rotate k so as to keep its window of compromise small. Moreover, if each key should be generated from a source of safe, cryptographic randomness.

A critical aspect of this protocol is reliance on [I-D.irtf-cfrg-hash-to-curve] for mapping arbitrary inputs x to points on a curve. Security requires this mapping be pre-image and collision resistant.

8.1. Timing Leaks

To ensure no information is leaked during protocol execution, all operations that use secret data MUST be constant time. Operations that SHOULD be constant time include: H_1() (hashing arbitrary strings to curves) and DLEQ_Generate(). [I-D.irtf-cfrg-hash-to-curve] describes various algorithms for constant-time implementations of H_1.

8.2. Hashing to curves

We choose different encodings in relation to the elliptic curve that is used, all methods are illuminated precisely in [I-D.irtf-cfrg-hash-to-curve]. In summary, we use the simplified Shallue-Woestijne-Ulas algorithm for hashing binary strings to the P-256 curve; the Icart algorithm for hashing binary strings to P384; the Elligator2 algorithm for hashing binary strings to CURVE25519 and CURVE448.

8.3. Verifiability (key consistency)

DLEQ proofs are essential to the protocol to allow V to check that P's designated private key was used in the computation. A side effect of this property is that it prevents P from using a unique key for select verifiers as a way of "tagging" them. If all verifiers expect use of a certain private key, e.g., by locating P's public key published from a trusted registry, then P cannot present unique keys to an individual verifier.

For this side effect to hold, P must also be prevented from using other techniques to manipulate their public key within the trusted registry to reduce client anonymity. For example, if P's public key is rotated too frequently then this may stratify the user base into small anonymity groups (those with VOPRF_Eval outputs taken from a given key epoch). In this case, it may become practical to link VOPRF sessions for a given user and thus compromise their privacy.

Similarly, if P can publish N public keys to a trusted registry then P may be able to control presentation of these keys in such a way that V is retroactively identified by V's key choice across multiple requests.

9. Applications

This section describes various applications of the VOPRF protocol.

9.1. Privacy Pass

This VOPRF protocol is used by the Privacy Pass system [PrivacyPass] to help Tor users bypass CAPTCHA challenges. Their system works as follows. Client C connects - through Tor - to an edge server E serving content. Upon receipt, E serves a CAPTCHA to C, who then solves the CAPTCHA and supplies, in response, n blinded points. E verifies the CAPTCHA response and, if valid, signs (at most) n blinded points, which are then returned to C along with a batched DLEQ proof. C stores the tokens if the batched proof verifies correctly. When C attempts to connect to E again and is prompted with a CAPTCHA, C uses one of the unblinded and signed points, or tokens, to derive a shared symmetric key sk used to MAC the CAPTCHA challenge. C sends the CAPTCHA, MAC, and token input x to E, who can use x to derive sk and verify the CAPTCHA MAC. Thus, each token is used at most once by the system.

The Privacy Pass implementation uses the P-256 instantiation of the VOPRF protocol. For more details, see [DGSTV18].

9.2. Private Password Checker

In this application, let D be a collection of plaintext passwords obtained by prover P. For each password p in D, P computes VOPRF_Eval on H_1(p), where H_1 is as described above, and stores the result in a separate collection D'. P then publishes D' with Y, its public key. If a client C wishes to query D' for a password p', it runs the VOPRF protocol using p as input x to obtain output y. By construction, y will be the OPRF evaluation of p hashed onto the curve. C can then search D' for y to determine if there is a match.

Concrete examples of important applications in the password domain include:

- password-protected storage [JKK14], [JKKX16];
- perfectly-hiding password management [SJKS17];
- password-protected secret-sharing [JKKX17].

9.2.1. Parameter Commitments

For some applications, it may be desirable for P to bind tokens to certain parameters, e.g., protocol versions, ciphersuites, etc. To accomplish this, P should use a distinct scalar for each parameter combination. Upon redemption of a token T from V, P can later verify that T was generated using the scalar associated with the corresponding parameters.

10. Acknowledgements

This document resulted from the work of the Privacy Pass team [PrivacyPass]. The authors would also like to acknowledge the helpful conversations with Hugo Krawczyk. Eli-Shaoul Khedouri provided additional review and comments on key consistency.

11. Normative References

[ChaumBlindSignature]
"Blind Signatures for Untraceable Payments", n.d., <http://sceweb.sce.uhcl.edu/yang/teaching/csci5234WebSecurityFall2011/Chaum-blind-signatures.PDF>.

[ChaumPedersen]
"Wallet Databases with Observers", n.d., <https://chaum.com/publications/Wallet_Databases.pdf>.

[DECAF] "Decaf, Eliminating cofactors through point compression", n.d., <https://www.shiftleft.org/papers/decaf/decaf.pdf>.

[DGSTV18] "Privacy Pass, Bypassing Internet Challenges Anonymously", n.d., <https://www.degruyter.com/view/j/popets.2018.2018.issue-3/popets-2018-0026/popets-2018-0026.xml>.

[I-D.irtf-cfrg-hash-to-curve]
Scott, S., Sullivan, N., and C. Wood, "Hashing to Elliptic Curves", draft-irtf-cfrg-hash-to-curve-03 (work in progress), March 2019.

[JKK14] "Round-Optimal Password-Protected Secret Sharing and T-PAKE in the Password-Only model", n.d., <https://eprint.iacr.org/2014/650>.

[JKKX16] "Highly-Efficient and Composable Password-Protected Secret Sharing (Or, How to Protect Your Bitcoin Wallet Online)", n.d., <https://eprint.iacr.org/2016/144>.

[JKKX17] "TOPPSS: Cost-minimal Password-Protected Secret Sharing based on Threshold OPRF", n.d., <https://eprint.iacr.org/2017/363>.

[NIST] "Keylength - NIST Report on Cryptographic Key Length and Cryptoperiod (2016)", n.d., <https://www.keylength.com/en/4/>.

[OPAQUE] "The OPAQUE Asymmetric PAKE Protocol", n.d., <https://tools.ietf.org/html/draft-krawczyk-cfrg-opaque-01>.

[PrivacyPass]
"Privacy Pass", n.d., <https://github.com/privacypass/challenge-bypass-server>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/info/rfc5869>.

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <https://www.rfc-editor.org/info/rfc7748>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/info/rfc8032>.

[RISTRETTO]
"The ristretto255 Group", n.d., <https://tools.ietf.org/html/draft-hdevalence-cfrg-ristretto-00>.

[SEC2] Standards for Efficient Cryptography Group (SECG), ., "SEC 2: Recommended Elliptic Curve Domain Parameters", n.d., <http://www.secg.org/sec2-v2.pdf>.

[SHAKE] "SHA-3 Standard, Permutation-Based Hash and Extendable-Output Functions", n.d., <https://www.nist.gov/publications/sha-3-standard-permutation-based-hash-and-extendable-output-functions?pub_id=919061>.

[SJKS17] "SPHINX, A Password Store that Perfectly Hides from Itself", n.d., <https://eprint.iacr.org/2018/695>.

Appendix A. Test Vectors

This section includes test vectors for the ECVOPRF-P256-HKDF-SHA256 VOPRF ciphersuite, including batched DLEQ output.

```
P-256
X: 04b14b08f954f5b6ab1d014b1398f03881d70842acdf06194eb96a6d08186f8cb985c1c5521
\
    f4ee19e290745331f7eb89a4053de0673dc8ef14cfe9bf8226c6b31
r: b72265c85b1ba42cfed7caaf00d2ccac0b1a99259ba0dbb5a1fc2941526a6849
M: 046025a41f81a160c648cfe8fdcaa42e5f7da7a71055f8e23f1dc7e4204ab84b705043ba5c7
\
    000123e1fd058150a4d3797008f57a8b2537766d9419c7396ba5279
k: f84e197c8b712cdf452d2cff52dec1bd96220ed7b9a6f66ed28c67503ae62133
Z: 043ab5ccb690d844dcb780b2d9e59126d62bc853ba01b2c339ba1c1b78c03e4b6adc5402f77
\
    9fc29f639edc138012f0e61960e1784973b37f864e4dc8abbc68e0b
N: 04e8aa6792d859075821e2fba28500d6974ba776fe230ba47ef7e42be1d967654ce776f889e
\
    e1f374ffa0bce904408aaa4ed8a19c6cc7801022b7848031f4e442a
D: { s: faddfaf6b5d6b4b6357adf856fc1e0044614ebf9dafdb4c6541c1c9e61243c5b,
     c: 8b403e170b56c915cc18864b3ab3c2502bd8f5ca25301bc03ab5138343040c7b }

P-256
X: 047e8d567e854e6bdc95727d48b40cbb5569299e0a4e339b6d707b2da3508eb6c238d3d4cb4
\
    68afc6ffc82fccbda8051478d1d2c9b21ffdfd628506c873ebb1249
r: f222dfe530fdbfcb02eb851867bfa8a6da1664dfc7cee4a51eb6ff83c901e15e
M: 04e2efdc73747e15e38b7a1bb90fe5e4ef964b3b8dccfda428f85a431420c84efca02f0f09c
\
    83a8241b44572a059ab49c080a39d0bce2d5d0b44ff5d012b5184e7
k: fb164de0a87e601fd4435c0d7441ff822b5fa5975d0c68035beac05a82c41118
Z: 049d01e1c555bd3324e8ce93a13946b98bdcc765298e6d60808f93c00bdfba2ebf48eef8f28
\
    d8c91c903ad6bea3d840f3b9631424a6cc543a0a0e1f2d487192d5b
N: 04723880e480b60b4415ca627585d1715ab5965570d30c94391a8b023f8854ac26f76c1d6ab
\
    bb38688a5affbcadad50ecbf7c93ef33ddfd735003b5a4b1a21ba14
D: { s: dfdf6ae40d141b61d5b2d72cf39c4a6c88db6ac5b12044a70c212e2bf80255b4,
     c: 271979a6b51d5f71719127102621fe250e3235867cfcf8dea749c3e253b81997 }

Batched DLEQ (P256)
M_0:
046025a41f81a160c648cfe8fdcaa42e5f7da7a71055f8e23f1dc7e4204ab84b705043ba5c\
    7000123e1fd058150a4d3797008f57a8b2537766d9419c7396ba5279
M_1:
04e2efdc73747e15e38b7a1bb90fe5e4ef964b3b8dccfda428f85a431420c84efca02f0f09\
    c83a8241b44572a059ab49c080a39d0bce2d5d0b44ff5d012b5184e7
```

```
Z_0:
043ab5ccb690d844dcb780b2d9e59126d62bc853ba01b2c339ba1c1b78c03e4b6adc5402f7\
    79fc29f639edc138012f0e61960e1784973b37f864e4dc8abbc68e0b
Z_1:
04647e1ab7946b10c1c1c92dd333e2fc9e93e85fdef5939bf2f376ae859248513e0cd91115\
    e48c6852d8dd173956aec7a81401c3f63a133934898d177f2a237eeb
k: f84e197c8b712cdf452d2cff52dec1bd96220ed7b9a6f66ed28c67503ae62133
H_5: HKDF-Expand-SHA256
label: "DLEQ_PROOF"
D: { s: b2123044e633d4721894d573decebc9366869fe3c6b4b79a00311ecfa46c9e34,
     c: 3506df9008e60130fcddf86fdb02cbfe4ceb88ff73f66953b1606f6603309862 }
```

Authors' Addresses

Alex Davidson
Cloudflare
County Hall
London, SE1 7GP
United Kingdom

Email: adavidson@cloudflare.com

Nick Sullivan
Cloudflare
101 Townsend St
San Francisco
United States of America

Email: nick@cloudflare.com

Christopher A. Wood
Apple Inc.
One Apple Park Way
Cupertino, California 95014
United States of America

Email: cawood@apple.com