Network Working Group
Internet-Draft
Intended status: Informational
Expires: September 10, 2020

A. Davidson
N. Sullivan
Cloudflare
C. Wood
Apple Inc.

March 09, 2020

Oblivious Pseudorandom Functions (OPRFs) using Prime-Order Groups
draft-irtf-cfrg-voprf-03

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. 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: Describe conventions and assumptions made relating to security of (V)OPRFs and prime-order group instantiations.

- 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 4.6 and for VOPRFs in Section 4.7.

- 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.

1.1. Change log

draft-03 [1]:

- Certify public key during VerifiableFinalize

- Remove protocol integration advice

- Add text discussing how to perform domain separation

- Drop OPRF_/VOPRF_ prefix from algorithm names

- Make prime-order group assumption explicit

- Changes to algorithms accepting batched inputs

- Changes to construction of batched DLEQ proofs

- Updated ciphersuites to be consistent with hash-to-curve and added OPRF specific ciphersuites

draft-02 [2]:

- Added section discussing cryptographic security and static DH oracles

- Updated batched proof algorithms

draft-01 [3]:

- Updated ciphersuites to be in line with https://tools.ietf.org/html/draft-irtf-cfrg-hash-to-curve-04

- Made some necessary modular reductions more explicit

1.2. Terminology

The following terms are used throughout this document.

- PRF: Pseudorandom Function.
- OPRF: Oblivious PRF.
- VOPRF: Verifiable Oblivious Pseudorandom Function.
- Verifier (V): Protocol initiator when computing F(k, x), also known as client.
- Prover (P): Holder of secret key k, also known as server.
- NIZK: Non-interactive zero knowledge.
- DLEQ: Discrete Logarithm Equality.

1.3. 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. Preliminaries

We start by detailing some necessary cryptographic definitions.

3.1. Security Properties

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

- 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.

3.2. Prime-order group instantiation

In this document, we assume the construction of a prime-order group GG for performing all mathematical operations. Such a group MUST provide the interface provided by cyclic group under the addition operation (for example, well-defined addition of group elements). We also assume the presence of a fixed generator G that can be detailed as a fixed parameter in the description of the group. We write p = order(GG) to represent the order of the group throughout this document.

It is common in cryptographic applications to instantiate such prime-order groups using elliptic curves, such as those detailed in [SEC2]. For some choices of elliptic curves (e.g. those detailed in [RFC7748] require accounting for cofactors) there are some implementation issues that introduce inherent discrepancies between standard prime-order groups and the elliptic curve instantiation. In this document, all algorithms that we detail assume that the group is a prime-order group, and this MUST be upheld by any implementer. That is, any curve instantiation should be written such that any discrepancies with a prime-order group instantiation are removed. In the case of cofactors, for example, this can be done by building cofactor multiplication into all elliptic curve operations.

3.3. Conventions

We detail a list of conventions that we use throughout this document.

3.3.1. Binary strings

- We use the notation x <-$ Q to denote sampling x from the uniform distribution over the set Q.
- We use x <- {0,1}^u to denote sampling x uniformly from the set of binary strings of length u. We may interpret x afterwards as a byte array.
- We say that x is a binary string of arbitrary-length (or alternatively sampled from {0,1}^*) if there is no fixed-size requirement on x.
- For two byte arrays x & y, write x .. y to denote their concatenation.

3.3.2. Group notation

- We use the letter p to denote the order of a group GG throughout, where the instantiation of the specific group is defined by context.
- For elements A & B of GG, we write A + B to denote the addition of thr group elements.
- We use GF(p) to denote the Galois Field of scalar values associated with the group GG.
- For a scalar r in GF(p), and a group element A, we write rA to denote the scalar multiplication of A.
- For two scalars r, s in GF(p), we use r+s to denote the resulting scalar in GF(p) (we may optionally write r+s mod p to make the modular reduction explicit).

4. OPRF Protocol

In this section we describe the OPRF and VOPRF protocols. Recall that such a protocol takes place between a verifier (V) and a prover (P). Commonly, V is a client and P is a server, and so we use these names interchangeably throughout. We always operate under the assumption that the verifier is a client, and the prover is a server in the interaction (and so we will use these names interchangeably throughout). The server holds a secret key k for a PRF. The protocol allows the client to learn PRF evaluations on chosen inputs x in such a way that the server learns nothing of x.

Our OPRF construction is based on the VOPRF construction known as 2HashDH-NIZK given by [JKK14]; essentially without providing zero-knowledge proofs that verify that the output is correct. Our VOPRF construction (including the NIZK DLEQ proofs from Section 5) is identical to the [JKK14] construction. With batched proofs (Section 6) our construction differs slightly in that we can perform multiple VOPRF evaluations in one go, whilst only constructing one NIZK proof object.

In this section we describe the OPRF and VOPRF protocols. Recall that such a protocol takes place between a verifier (V) and a prover (P). We may commonly think of the verifier as the client, and the prover as the server in the interaction (we will use these names interchangeably throughout). The server holds a key k for a PRF. The protocol allows the client to learn PRF evaluations on chosen inputs x without revealing x to the server.

Our OPRF construction is based on the VOPRF construction known as 2HashDH-NIZK given by [JKK14]; essentially without providing zero-knowledge proofs that verify that the output is correct. Our VOPRF construction (including the NIZK DLEQ proofs from Section 5) is identical to the [JKK14] construction. With batched proofs (Section 6) our construction differs slightly in that we can perform multiple VOPRF evaluations in one go, whilst only constructing one NIZK proof object.

4.1. Design

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 (H_1: {0,1}^* -> GG) and H_2 maps two arbitrary inputs to a fixed-length (w) output (H_2: {0,1}^u x {0,1}^v -> {0,1}^w), e.g., HMAC_SHA256. All hash functions in the protocol are modeled as random oracles. Let L be the security parameter. Let k be the prover's 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 binary string that is the verifier's input to the OPRF protocol (this can be of arbitrary length).

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 (along with a domain separating label DST) using H_2 to yield an output. This flow is illustrated below.

```
     Verifier(x)                    Prover(k)
  ------------------------------------------------------------
     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(DST, 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(DST, x .. N). Note that the output from P is not the PRF value because the actual input x is blinded by r.

The security of our construction is discussed in more detail in Section 8.1.2.

4.2. Protocol functionality

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

- Setup(l): Let GG=GG(l) be a group with a prime-order p=p(l) (e.g., p is l-bits long). Randomly sample an integer k in GF(p) and output (k,GG)

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

- Evaluate(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.

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

- Finalize(x,N,aux?): Finalize N by first computing dk := H_2(DST, x .. N). Subsequently output y := H_2(dk, aux), where aux is some auxiliary data encoded as a byte string. If aux is not specified, it defaults to the empty byte string.

For verifiability (VOPRF) we modify the algorithms of VerifiableSetup, VerifiableEvaluate and VerifiableUnblind to be the following:

- VerifiableSetup(l): Run (k,GG) = Setup(l), compute Y = kG, where G is a generator of the group GG. Output (k,GG,Y).

- VerifiableEvaluate(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 Evaluate.

- VerifiableUnblind(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".

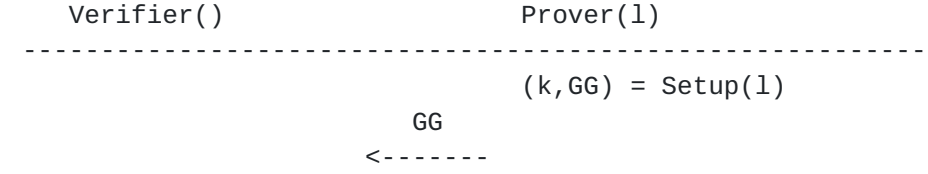
- VerifiableFinalize(x,Y,N,aux?): Same as Finalize, except we now compute dk := H_2(DST, x .. Y .. N), i.e. we also certify the public key in the finalization process.

We leave the rest of the OPRF algorithms unmodified. When referring explicitly to VOPRF execution, we replace 'OPRF' in all method names with 'VOPRF'. We describe explicit instantiations of these functions in Section 4.6 and Section 4.7.

4.2.1. Generalized OPRF

Using the API provided by the functions above, we can restate the OPRF protocol using the following descriptions. The first protocol refers to the OPRF setup phase that is run by the server. This generates the secret input used by the server and the public information that is given to the client.

OPRF setup phase:



OPRF evaluation phase:

```
   Verifier(x,aux)                  Prover(k)
-----------------------------------------------------------
   (r, M) = Blind(x)
                          M
                       ------->
                                Z = Evaluate(k,M)
                          Z
                       <-------
   N = Unblind(r,Z)
   Output Finalize(x,N,aux)
```

Note that in the final output, the client computes Finalize over some auxiliary input data aux.

4.2.2. Generalized VOPRF

The generalized VOPRF functionality differs slightly from the OPRF protocol above. Firstly, the server sends over an extra commitment value Y = kG, where G is a common generator known to both participants. Secondly, the server sends over both outputs from VerifiableEvaluate in the evaluation phase, and the client also verifies the server's output.

VOPRF setup phase:

```
   Verifier()                   Prover(l)
-----------------------------------------------------------
                                (k,GG,Y) = VerifiableSetup(l)
                        (GG,Y)
                       <-------
```

VOPRF evaluation phase:

```
   Verifier(x,Y,aux)            Prover(k)
-----------------------------------------------------------
   (r, M) = VerifiableBlind(x)
                          M
                       ------->
                                (Z,D) = VerifiableEvaluate(k,G,Y,M)
                          (Z,D)
                       <-------
   N = VerifiableUnblind(r,G,Y,M,Z,D)
   Output VerifiableFinalize(x,Y,N,aux)
```

4.3. Protocol correctness

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

```
Finalize(x, Unblind(r,M,Evaluate(k,M)), aux)
    == H_2(H_2(DST, x .. F(k,x)), aux)
```

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

```
Z = VerifiableEvaluate(k,G,Y,M)
VerifiableFinalize(x, Y, VerifiableUnblind(r,G,Y,M,Z), aux)
    == H_2(H_2(DST, x .. F(k,x)), aux)
```

with overwhelming probability, where (r, M) = VerifiableBlind(x). In other words, the inner H_2 invocation effectively derives a key, dk, from the input data DST, x, N. The outer invocation derives the output y by evaluating H_2 over dk and auxiliary data aux.

4.4. Domain separation

The Finalize procedure accepts optional auxiliary byte string input (aux) as a means of modifying the PRF output. This parameter SHOULD be used for domain separation in (V)OPRF the protocol. Specifically, any system which has multiple (V)OPRF applications should use separate aux values to to ensure finalized outputs are separate. Guidance for constructing aux can be found in [I-D.irtf-cfrg-hash-to-curve]; Section 3.1.

4.5. Instantiations of GG

As we remarked above, GG is a group 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 available to both the client and the server ahead of the protocol, or derived for each different group instantiation using a fixed method. In the elliptic curve setting, we recommend using the fixed generators that are given as part of the curve description.

4.6. OPRF algorithms

This section provides descriptions of the algorithms used in the generalized protocols from Section 4.2.1. We describe the VOPRF analogues for the protocols in Section 4.2.2 later in Section 4.7.

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 4.8.

4.6.1. Setup

Input:

l: Some suitable choice of prime length for instantiating a group structure (e.g. as described in [NIST]).

Output:

k: A key chosen from {0,1}^l and interpreted as a scalar in [1,p-1].
GG: A cyclic group with prime-order p of length l bits.

Steps:

1. Construct a group GG = GG(l) with prime-order p of length l bits
2. k <-$ GF(p)
3. Output (k,GG)

4.6.2. Blind

Input:

x: Binary string taken from {0,1}^*.

Output:

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

Steps:

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

4.6.3. Evaluate

Input:

k: A scalar value taken from [1,p-1].
M: An element in GG.

Output:

Z: An element in GG.

Steps:

1. Z := kM
2. Output Z

4.6.4. Unblind

Input:

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

Output:

N: An element in GG.

Steps:

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

4.6.5. Finalize

Input:

x: Binary string taken from {0,1}^*.
N: An element in GG.
aux: Arbitrary auxiliary data (as bytes).

Output:

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

Steps:

1. DST := "oprf_derive_output"
2. dk := H_2(DST, x .. N)
3. y := H_2(dk, aux)
4. Output y

4.7. VOPRF algorithms

We make modifications to the aforementioned algorithms in the VOPRF setting.

4.7.1. VerifiableSetup

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 a scalar in [1,p-1].
GG: A cyclic group with prime-order p of length l bits.
Y: A group element in GG.

Steps:

1. (k,GG) <- Setup(l)
2. Y := kG
3. Output (k,GG,Y)

4.7.2. VerifiableBlind

Input:

x: V's PRF input.

Output:

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

Steps:

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

4.7.3. VerifiableEvaluate

Input:

k: A random scalar in [1,p-1].
G: Public fixed generator of group GG.
Y: An element in GG.
M: An element in GG.

Output:

Z: An 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)

4.7.4. VerifiableUnblind

Input:

r: Random scalar in [1, p - 1].
G: Public fixed generator of group GG.
Y: An element in GG.
M: An element in GG.
Z: An element in GG.
D: DLEQ proof object.

Output:

N: An element in GG.

Steps:

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

4.7.5. VerifiableFinalize

Input:

x: Binary string in {0,1}^*.
Y: An element in GG.
N: An element in GG, or "error".
aux: Arbitrary auxiliary data in {0,1}^*.

Output:

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

Steps:

1. If N == "error", output "error".
2. DST := "voprf_derive_output"
3. dk := H_2(DST, x .. Y .. N)
4. y := H_2(dk, aux)
5. Output y

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

In Section Section 4.6 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 Blind and Unblind (and similarly for the VOPRF variants).

We define the FBB variants of the algorithms in Section 4.6 below along with a new algorithm 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.

4.8.1. 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)

4.8.2. Blind

Input:

x: Binary string in {0,1}^*.
rG: An element in GG.

Output:

M: An element in GG.

Steps:

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

4.8.3. Unblind

Input:

rY: An element in GG.
M: An element in GG.
Z: An element in GG.

Output:

N: An element in GG.

Steps:

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

Notice that 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 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 VerifiableSetup(l) satisfies log_G(Y) == log_M(Z) where Z is the output of VerifiableEvaluate(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, modeled as a random oracle.

Output:

D: DLEQ proof (c, s).

Steps:

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

6. Batched VOPRF evaluation

Common applications (e.g. [PrivacyPass]) require V to obtain multiple PRF evaluations from P. In the VOPRF case, this would naively require running multiple protocol invocations. This is costly, both in terms of computation and communication. To get around this, applications can 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.

Since P is the VOPRF server, it may be able to 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 two additional hash functions H_4: GG^(2n+2) -> {0,1}^a, and H_5: {0,1}^a x ZZ^3 -> {0,1}^b (both modeled as random oracles).

We can instantiate the random oracle function H_4 using the same hash function that is used for H_3 previously. 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]. This is preferable in situations where we may require outputs that are larger than 512 bits in length, for example.

6.1. Batched_DLEQ_Generate

Input:

k: Evaluator secret key.
G: Public fixed generator of group GG (with order p).
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 random oracle hash function from GG^(2n+2) to {0,1}^a.
H_5: A random oracle hash function from {0,1}^a x ZZ^2 to {0,1}^b.
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. i' := i
3. for i in [m]:
   1. di <- H_5(seed,i',info)
   2. if di > p:
      1. i' = i'+1
      2. i = i-1 // decrement and try again
      3. continue
4. c1,...,cn := (int)d1,...,(int)dn
5. M := c1M1 + ... + cnMn
6. Z := c1Z1 + ... + cnZn
7. Output DLEQ_Generate(k,G,Y,M,Z)

6.2. DLEQ_Batched_Verify

```
Input:

 G: Public fixed generator of group GG (with order p).
 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. i' := i
 3. for i in [m]:
    1. di <- H_5(seed,i',info)
    2. if di > p:
       1. i' = i'+1
       2. i = i-1 // decrement and try again
       3. continue
 4. c1,...,cn := (int)d1,...,(int)dn
 5. M := c1M1 + ... + cnMn
 6. Z := c1Z1 + ... + cnZn
 7. Output DLEQ_Verify(G,Y,M,Z,D)
```

6.3. Modified algorithms

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 DLEQ_Batched_Generate over these arrays. Concretely, we modify the following algorithms:

6.3.1. VerifiableBlind

Input:

[xi]: An array of m binary strings taken from {0,1}^*.

Output:

[ri]: An array of m random scalars in [1, p - 1].
[Mi]: An array of elements in GG.

Steps:

1. groupElems = []
2. blinds = []
3. for i in [m]:
    1. ri <-$ GF(p)
    2. Mi := rH_1(xi)
    3. blinds.push(ri)
    4. groupElems.push(Mi)
4. Output (blinds, groupElems)

6.3.2. VerifiableEvaluate

Input:

k: Evaluator secret key.
G: Public fixed generator of group GG.
Y: Evaluator public key (= kG).
[Mi]: An array of m elements in GG.

Output:

[Zi]: An array of m elements in GG.
D: Batched DLEQ proof object.

Steps:

1. outputElems = []
2. for i in [m]:
    1. Zi := kMi
    2. outputElems.push(Zi)
3. D = Batched_DLEQ_Generate(k,G,Y,[Mi],outputElems)
4. Output (outputElems, D)

6.3.3. VerifiableUnblind

Input:

```
G:      Public fixed generator of group GG.
Y:      Evaluator public key (= kG).
[ Mi ]: An array of m elements in GG.
[ Zi ]: An array of m elements in GG.
[ ri ]: An array of m random scalars in [1, p - 1].
D:      Batched DLEQ proof object.
```

Output:

```
[ Ni ]: An array of n elements in GG.
```

Steps:

1. if !Batch_DLEQ_Verify(G,Y,[Mi],[Zi],D): Output "error"
2. N = []
3. for i in [m]:
    1. Ni := (ri^(-1))Zi
    2. N.push(Ni)
4. Output N

6.3.4. VerifiableFinalize

The description of this algorithm does not change in the batched case. Instead, the protocol description in Section 4.2.2 changes so that "VerifiableFinalize" runs once for each of the outputs of "VerifiableUnblind".

6.4. 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].

Input:

[ri]: Random scalars in [1, p - 1].
G: Public fixed generator of group GG.
Y: Evaluator public key.
[Mi]: Blinded elements of GG.
[Zi]: Server-generated elements in GG.
D: A batched DLEQ proof object.

Output:

N: element in GG, or "error".

Steps:

1. N := (r^(-1))Z
2. If 1 = DLEQ_Batched_Verify(G,Y,[Mi],[Zi],D), output N
3. Output "error"

7. Supported ciphersuites

This section specifies supported VOPRF 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 alone can be allowed by simply dropping the relevant components. For reasons that are detailed in Section 8.1, we only consider ciphersuites that provide strictly greater than 128 bits of security [NIST].

7.1. OPRF-curve448-HKDF-SHA512-ELL2-RO:

- GG: curve448 [RFC7748]
- H_1: curve448-SHA512-ELL2-RO [I-D.irtf-cfrg-hash-to-curve]
  - hash-to-curve DST: "RFCXXXX-OPRF-curve448-SHA512-ELL2-RO-"
- H_2: HMAC_SHA512 [RFC2104]
- H_3: SHA512

7.2. OPRF-P384-HKDF-SHA512-SSWU-RO:

- GG: secp384r1 [SEC2]
- H_1: P384-SHA512-SSWU-RO [I-D.irtf-cfrg-hash-to-curve]

  * hash-to-curve DST: "RFCXXXX-OPRF-P384-SHA512-SSWU-RO-"

- H_2: HMAC_SHA512 [RFC2104]
- H_3: SHA512

7.3. OPRF-P521-HKDF-SHA512-SSWU-RO:

- GG: secp521r1 [SEC2]
- H_1: P521-SHA512-SSWU-RO [I-D.irtf-cfrg-hash-to-curve]
  * hash-to-curve DST: "RFCXXXX-OPRF-P521-SHA512-SSWU-RO-"
- H_2: HMAC_SHA512 [RFC2104]
- H_3: SHA512

7.4. VOPRF-curve448-HKDF-SHA512-ELL2-RO:

- GG: curve448 [RFC7748]
- H_1: curve448-SHA512-ELL2-RO [I-D.irtf-cfrg-hash-to-curve]
  * hash-to-curve DST: "RFCXXXX-VOPRF-curve448-SHA512-ELL2-RO-"
- H_2: HMAC_SHA512 [RFC2104]
- H_3: SHA512
- H_4: SHA512
- H_5: HKDF-Expand-SHA512

7.5. VOPRF-P384-HKDF-SHA512-SSWU-RO:

- GG: secp384r1 [SEC2]
- H_1: P384-SHA512-SSWU-RO [I-D.irtf-cfrg-hash-to-curve]
  * hash-to-curve DST: "RFCXXXX-VOPRF-P384-SHA512-SSWU-RO-"
- H_2: HMAC_SHA512 [RFC2104]
- H_3: SHA512
- H_4: SHA512

- H_5: HKDF-Expand-SHA512

7.6. VOPRF-P521-HKDF-SHA512-SSWU-RO:

- GG: secp521r1 [SEC2]
- H_1: P521-SHA512-SSWU-RO [I-D.irtf-cfrg-hash-to-curve]
  - hash-to-curve DST: "RFCXXXX-VOPRF-P521-SHA512-SSWU-RO-"
- H_2: HMAC_SHA512 [RFC2104]
- H_3: SHA512
- H_4: SHA512
- H_5: HKDF-Expand-SHA512

We remark that the 'hash-to-curve DST' field is necessary for domain separation of the hash-to-curve functionality.

8. Security Considerations

This section discusses the cryptographic security of our protocol, along with some suggestions and trade-offs that arise from the implementation of the implementation of an OPRF.

8.1. Cryptographic security

We discuss the cryptographic security of the OPRF protocol from Section 4, relative to the necessary cryptographic assumptions that need to be made.

8.1.1. Computational hardness assumptions

Each assumption states that the problems specified below are computationally difficult to solve in relation to sp (the security parameter). In other words, the probability that an adversary has in solving the problem is bounded by a function negl(sp), where negl(sp) < 1/f(sp) for all polynomial functions f().

Let GG = GG(sp) be a group with prime-order p, and let FFp be the finite field of order p.

8.1.1.1. Discrete-log (DL) problem

Given G, a generator of GG, and H = hG for some h in FFp; output h.

8.1.1.2. Decisional Diffie-Hellman (DDH) problem

Sample a uniformly random bit d in {0,1}. Given (G, aG, bG, C), where:

- G is a generator of GG;
- a,b are elements of FFp;
- if d == 0: C = abG; else: C is sampled uniformly GG(sp).

Output d' == d.

8.1.2. Protocol security

As aforementioned, our OPRF and VOPRF constructions are based heavily on the 2HashDH-NIZK construction given in [JKK14], except for considerations on how we instantiate the NIZK DLEQ proof system. This means that the cryptographic security of our construction is also based on the assumption that the One-More Gap DH is computationally difficult to solve.

The (N,Q)-One-More Gap DH (OMDH) problem asks the following.

> Given:
> - G, kG, G_1, ... , G_N where G, G1, ... GN are elements od GG;
> - oracle access to an OPRF functionality using the key k;
> - oracle access to DDH solvers.
>
> Find Q+1 pairs of the form below:
>
> (G_{j_s}, kG_{j_s})
>
> where the following conditions hold:
> - s is a number between 1 and Q+1;
> - j_s is a number between 1 and N for each s;
> - Q is the number of allowed queries.

The original paper [JKK14] gives a security proof that the 2HashDH-NIZK construction satisfies the security guarantees of a VOPRF protocol Section 3.1 under the OMDH assumption in the universal composability (UC) security model. Without the NIZK proof system, the protocol instantiates an OPRF protocol only. See the paper for further details.

8.1.3. Q-strong-DH oracle

A side-effect of our OPRF design is that it allows instantiation of a oracle for constructing Q-strong-DH (Q-sDH) samples. The Q-Strong-DH problem asks the following.

> Given G1, G2, h*G2, (h^2)*G2, ..., (h^Q)*G2; for G1 and G2 generators of GG.
>
> Output ((1/(k+c))*G1, c) where c is an element of FFp

The assumption that this problem is hard was first introduced in [BB04]. Since then, there have been a number of cryptanalytic studies that have reduced the security of the assumption below that implied by the group instantiation (for example, [BG04] and [Cheon06]). In summary, the attacks reduce the security of the group instantiation by log_2(Q) bits.

As an example, suppose that a group instantiation is used that provides 128 bits of security. Then an adversary with access to a Q-sDH oracle and makes Q=2^20 queries can reduce the security of the instantiation by log_2(2^20) = 20 bits.

Notice that it is easy to instantiate a Q-sDH oracle using the OPRF functionality that we provide. A client can just submit sequential queries of the form (G, kG, (k^2)G, ..., (k^(Q-1))G), where each query is the output of the previous interaction. This means that any client that submit Q queries to the OPRF can use the aforementioned attacks to reduce security of the group instantiation by log_2(Q) bits.

Recall that from a malicious client's perspective, the adversary wins if they can distinguish the OPRF interaction from a protocol that computes the ideal functionality provided by the PRF.

8.1.4. Implications for ciphersuite choices

The OPRF instantiations that we recommend in this document are informed by the cryptanalytic discussion above. In particular, choosing elliptic curves configurations that describe 128-bit group instantiations would appear to in fact instantiate an OPRF with 128-log_2(Q) bits of security.

While it would require an informed and persistent attacker to launch a highly expensive attack to reduce security to anything much below 100 bits of security, we see this possibility as something that may result in problems in the future. Therefore, all of our ciphersuites in Section 7 come with a minimum group instantiation corresponding to

196 bits of security. This would require an adversary to launch a minimum of Q = 2^(68) queries to reduce security to 128 bits using the Q-sDH attacks. As a result, it appears prohibitively expensive to launch credible attacks on these parameters with our current understanding of the attack surface.

8.2. Hashing to curve

A critical aspect of implementing this protocol using elliptic curve group instantiations is a method of instantiating the function H1, that maps inputs to group elements. In the elliptic curve setting, this must be a deterministic function that maps arbitrary inputs x (as bytes) to uniformly chosen points in the curve.

In the security proof of the construction H1 is modeled as a random oracle. This implies that any instantiation of H1 must be pre-image and collision resistant. In Section 7 we give instantiations of this functionality based on the functions described in [I-D.irtf-cfrg-hash-to-curve]. Consequently, any OPRF implementation must adhere to the implementation and security considerations discussed in [I-D.irtf-cfrg-hash-to-curve] when instantiating the function H1.

8.3. 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(). As mentioned previously, [I-D.irtf-cfrg-hash-to-curve] describes various algorithms for constant-time implementations of H_1.

8.4. User segregation

The aim of the OPRF functionality is to allow clients receive pseudorandom function evaluations on their own inputs, without compromising their own privacy with respect to the server. In many applications (for example, [PrivacyPass]) the client may choose to reveal their original input, after an invocation of the OPRF protocol, along with their OPRF output. This can prove to the server that it has received a valid OPRF output in the past. Since the server does not reveal learn anything about the OPRF output, it should not be able to link the client to any previous protocol instantiation.

Consider a malicious server that manages to segregate the user base into different sets. Then this reduces the effective privacy of all of the clients involved, since the client above belongs to a smaller

set of users than previously hoped. In general, if the user-base of the OPRF functionality is quite small, then the obliviousness of clients is limited. That is, smaller user-bases mean that the server is able to identify client's with higher certainty.

In summary, an OPRF instantiation effectively comes with an additional privacy parameter pp. If all clients of the OPRF make one query and then subsequently reveal their OPRF input afterwards, then the server should be link the revealed input to a protocol instantiation with probability 1/pp.

Below, we provide a few techniques that could be used to abuse client-privacy in the OPRF construction by segregating the user-base, along with some mitigations.

8.4.1. Linkage patterns

If the server is able to ascertain patterns of usage for some clients - such as timings associated with usage - then the effective privacy of the clients is reduced to the number of users that fit each usage pattern. Along with early registration patterns, where early adopters initially have less privacy due to a low number of registered users, such problems are inherent to any anonymity-preserving system.

8.4.2. Evaluation on multiple keys

Such an attack consists of the server evaluating the OPRF on multiple different keys related to the number of clients that use the functionality. As an extreme, the server could evaluate the OPRF with a different key for each client. If the client then revealed their hidden information at a later date then the server would immediately know which initial request they launched.

The VOPRF variant helps mitigate this attack since each server evaluation can be bound to a known public key. However, there are still ways that the VOPRF construction can be abused. In particular:

- If the server successfully provisions a large number of keys that are trusted by clients, then the server can divide the user-base by the number of keys that are currently in use. As such, clients should only trust a small number (2 or 3 ideally) of server keys at any one time. Additionally, a tamper-proof audit log system akin to existing work on Key Transparency [keytrans] could be used to ensure that a server is abiding by the key policy. This would force the server to be held accountable for their key updates, and thus higher key update frequencies can be better managed on the client-side.

- If the server rotates their key frequently, then this may result in client's holding out-of-date information from a past interaction. Such information can also be used to segregate the user-base based on the last time that they accessed the OPRF protocol. Similarly to the above, server key rotations must be kept to relatively infrequent intervals (such as once per month). This will prevent too many clients from being segregated into different groups related to the time that they accessed the functionality. There are viable reasons for rotating the server key (for protecting against malicious clients) that we address more closely in Section 8.5.

Since key provisioning requires careful handling, all public keys should be accessible from a client-trusted registry with a way of auditing the history of key updates. We also recommend that public keys have a corresponding expiry date that clients can use to prevent the server from using keys that have been provisioned for a long period of time.

8.5. Key rotation

Since the server's key is critical to security, the longer it is exposed by performing (V)OPRF operations on client inputs, the longer it is possible that the key can be compromised. For instance, if the key is kept in production for a long period of time, then this may grant the client the ability to hoard large numbers of tokens. This has negative impacts for some of the applications that we consider in Section 9. As another example, if the key is kept in circulation for a long period of time, then it also allows the clients to make enough queries to launch more powerful variants of the Q-sDH attacks from Section 8.1.3.

To combat attacks of this nature, regular key rotation should be employed on the server-side. A suitable key-cycle for a key used to compute (V)OPRF evaluations would be between one week and six months.

As we discussed in Section 8.4.2, key rotation cycles that are too frequent (in the order of days) can lead to large segregation of the wider user base. As such, the length of the key cycles represent a trade-off between greater server key security (for shorter cycles), and better client privacy (for longer cycles). In situations where client privacy is paramount, longer key cycles should be employed. Otherwise, shorter key cycles can be managed if the server uses a Key Transparency-type system [keytrans]; this allows clients to publicly audit their rotations.

9. Applications

This section describes various applications of the (V)OPRF 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 VerifiableEvaluate 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. Contributors

- Alex Davidson (alex.davidson92@gmail.com)
- Nick Sullivan (nick@cloudflare.com)
- Chris Wood (cawood@apple.com)
- Eli-Shaoul Khedouri (eli@intuitionmachines.com)

11. 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.

12. References

12.1. Normative References

[BB04] "Short Signatures Without Random Oracles", <http://ai.stanford.edu/~xb/eurocrypt04a/bbsigs.pdf>.

[BG04] "The Static Diffie-Hellman Problem", <https://eprint.iacr.org/2004/306>.

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

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

[Cheon06] "Security Analysis of the Strong Diffie-Hellman Problem", <https://www.iacr.org/archive/eurocrypt2006/40040001/40040001.pdf>.

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

[DGSTV18] "Privacy Pass, Bypassing Internet Challenges Anonymously", <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] Faz-Hernandez, A., Scott, S., Sullivan, N., Wahby, R., and C. Wood, "Hashing to Elliptic Curves", draft-irtf-cfrg-hash-to-curve-05 (work in progress), November 2019.

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

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

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

[keytrans] "Security Through Transparency", <https://security.googleblog.com/2017/01/security-through-transparency.html>.

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

[OPAQUE] "The OPAQUE Asymmetric PAKE Protocol", <https://tools.ietf.org/html/draft-krawczyk-cfrg-opaque-02>.

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

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <https://www.rfc-editor.org/info/rfc2104>.

[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>.

[RISTRETTO] "The ristretto255 Group", <https://tools.ietf.org/html/draft-hdevalence-cfrg-ristretto-01>.

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

[SHAKE] "SHA-3 Standard, Permutation-Based Hash and Extendable-Output Functions", <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", <https://eprint.iacr.org/2018/695>.

12.2. URIs

[1] https://tools.ietf.org/html/draft-irtf-cfrg-voprf-03

[2] https://tools.ietf.org/html/draft-irtf-cfrg-voprf-02

[3] https://tools.ietf.org/html/draft-irtf-cfrg-voprf-01

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