Workgroup: Network Working Group
Internet-Draft: draft-irtf-cfrg-voprf-07
Published: 6 July 2021
Intended Status: Informational
Expires: 7 January 2022
Authors: A. Davidson
Brave Software
A. Faz-Hernandez
Cloudflare
N. Sullivan
Cloudflare
C.A. Wood
Cloudflare

Oblivious Pseudorandom Functions (OPRFs) using Prime-Order Groups

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.

Discussion Venues

This note is to be removed before publishing as an RFC.

Source for this draft and an issue tracker can be found at https://github.com/cfrg/draft-irtf-cfrg-voprf.

Status of This Memo

Table of Contents

1. Introduction

A pseudorandom function (PRF) F(k, x) is an efficiently computable function taking a private key k and a value x as input. This function is pseudorandom if the keyed function K(_) = F(K, _) is indistinguishable from a randomly sampled function acting on the same domain and range as K(). An oblivious PRF (OPRF) is a two-party protocol between a server and a client, where the server holds a PRF key k and the client holds some input x. The protocol allows both parties to cooperate in computing F(k, x) such that: the client learns F(k, x) without learning anything about k; and the server does not learn anything about x or F(k, x). A Verifiable OPRF (VOPRF) is an OPRF wherein the server can prove to the client that F(k, x) was computed using the key k.

The usage of OPRFs has been demonstrated in constructing a number of applications: password-protected secret sharing schemes [JKKX16]; privacy-preserving password stores [SJKS17]; and password-authenticated key exchange or PAKE [I-D.irtf-cfrg-opaque]. A VOPRF is necessary in some applications, e.g., the Privacy Pass protocol [I-D.davidson-pp-protocol], 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. [JKK14].

This document introduces an OPRF protocol built in prime-order groups, applying to finite fields of prime-order and also elliptic curve (EC) groups. 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, the public-facing API, and its security properties.

1.1. Change log

draft-07:

- Bind blinding mechanism to mode (additive for verifiable mode and multiplicative for base mode).
- Add explicit errors for deserialization.
- Document explicit errors and API considerations.
- Adopt SHAKE-256 for decaf448 ciphersuite.
- Normalize HashToScalar functionality for all ciphersuites.
- Refactor and generalize DLEQ proof functionality and domain separation tags for use in other protocols.
- Update test vectors.
- Apply various editorial changes.

draft-06:

- Specify of group element and scalar serialization.
- Remove info parameter from the protocol API and update domain separation guidance.
- Fold Unblind function into Finalize.
- Optimize ComputeComposites for servers (using knowledge of the private key).
- Specify deterministic key generation method.
- Update test vectors.
- Apply various editorial changes.

draft-05:

* Move to ristretto255 and decaf448 ciphersuites.

* Clean up ciphersuite definitions.

* Pin domain separation tag construction to draft version.

* Move key generation outside of context construction functions.

* Editorial changes.

draft-04:

* Introduce Client and Server contexts for controlling verifiability and required functionality.

* Condense API.

* Remove batching from standard functionality (included as an extension)

* Add Curve25519 and P-256 ciphersuites for applications that prevent strong-DH oracle attacks.

* Provide explicit prime-order group API and instantiation advice for each ciphersuite.

* Proof-of-concept implementation in sage.

* Remove privacy considerations advice as this depends on applications.

draft-03:

* 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:

* Added section discussing cryptographic security and static DH oracles.

* Updated batched proof algorithms.

draft-01:

* 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. Requirements

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. Preliminaries

The (V)OPRF protocol in this document has two primary dependencies:

* GG: A prime-order group implementing the API described below in Section 2.1, with base point defined in the corresponding reference for each group. (See Section 4 for these base points.)

* Hash: A cryptographic hash function that is indifferentiable from a Random Oracle, whose output length is Nh bytes long.

Section 4 specifies ciphersuites as combinations of GG and Hash.

2.1. Prime-Order Group Dependency

In this document, we assume the construction of an additive, prime-order group GG for performing all mathematical operations. Such groups are uniquely determined by the choice of the prime p that defines the order of the group. We use GF(p) to represent the finite field of order p. For the purpose of understanding and implementing this document, we take GF(p) to be equal to the set of integers defined by {0, 1, ..., p-1}.

The fundamental group operation is addition + with identity element I. For any elements A and B of the group GG, A + B = B + A is also a member of GG. Also, for any A in GG, there exists an element -A such that A + (-A) = (-A) + A = I. Scalar multiplication is equivalent to the repeated application of the group operation on an element A with

itself r-1 times, this is denoted as r*A = A + ... + A. For any element A, p*A=I. We denote G as the fixed generator of the group. Scalar base multiplication is equivalent to the repeated application of the group operation G with itself r-1 times, this is denoted as ScalarBaseMult(r). The set of scalars corresponds to GF(p). This document uses types Element and Scalar to denote elements of the group GG and its set of scalars, respectively.

We now detail a number of member functions that can be invoked on a prime-order group GG.

- *Order(): Outputs the order of GG (i.e. p).

- *Identity(): Outputs the identity element of the group (i.e. I).

- *HashToGroup(x): A member function of GG that deterministically maps an array of bytes x to an element of GG. The map must ensure that, for any adversary receiving R = HashToGroup(x), it is computationally difficult to reverse the mapping. This function is optionally parameterized by a domain separation tag (DST); see Section 4.

- *HashToScalar(x): A member function of GG that deterministically maps an array of bytes x to an element in GF(p). This function is optionally parameterized by a DST; see Section 4.

- *RandomScalar(): A member function of GG that chooses at random a non-zero element in GF(p).

- *SerializeElement(A): A member function of GG that maps a group element A to a unique byte array buf of fixed length Ne. The output type of this function is SerializedElement.

- *DeserializeElement(buf): A member function of GG that maps a byte array buf to a group element A, or fails if the input is not a valid byte representation of an element. This function can raise a DeserializeError if deserialization fails or A is the identity element of the group; see Section 6.4.

- *SerializeScalar(s): A member function of GG that maps a scalar element s to a unique byte array buf of fixed length Ns. The output type of this function is SerializedScalar.

- *DeserializeScalar(buf): A member function of GG that maps a byte array buf to a scalar s, or fails if the input is not a valid byte representation of a scalar. This function can raise a DeserializeError if deserialization fails; see Section 6.4.

Two functions can be used for generating a (V)OPRF key pair (skS, pkS) where skS is a non-zero integer less than p and pkS =

ScalarBaseMult(skS): GenerateKeyPair and DeriveKeyPair. GenerateKeyPair is a randomized function that outputs a fresh key pair (skS, pkS) upon ever invocation. DeriveKeyPair is a deterministic function that generates private key skS from a random byte string seed, which SHOULD have at least Ns bytes of entropy, and then computes pkS = ScalarBaseMult(skS).

It is convenient 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], which 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 implementation. That is, any curve instantiation should be written such that any discrepancies with a prime-order group instantiation are removed. See Section 4 for advice corresponding to the implementation of this interface for specific definitions of elliptic curves.

2.2. Conventions and Terminology

The following conventions are used throughout the document.

- For any object x, we write len(x) to denote its length in bytes.
- For two byte arrays x and y, write x || y to denote their concatenation.
- I2OSP and OS2IP: Convert a byte array to and from a non-negative integer as described in [RFC8017]. Note that these functions operate on byte arrays in big-endian byte order.

Data structure descriptions use TLS notation [RFC8446], Section 3.

All algorithm descriptions are written in a Python-like pseudocode. We also use the CT_EQUAL(a, b) function to represent constant-time byte-wise equality between byte arrays a and b. This function returns true if a and b are equal, and false otherwise.

The following terms are used throughout this document.

- PRF: Pseudorandom Function.
- OPRF: Oblivious Pseudorandom Function.
- VOPRF: Verifiable Oblivious Pseudorandom Function.

*Client: Protocol initiator. Learns pseudorandom function evaluation as the output of the protocol.

*Server: Computes the pseudorandom function over a secret key. Learns nothing about the client's input.

*NIZK: Non-interactive zero knowledge.

*DLEQ: Discrete Logarithm Equality.

3. (V)OPRF Protocol

In this section, we define two OPRF variants: a base mode and verifiable mode. In the base mode, a client and server interact to compute y = F(skS, x), where x is the client's input, skS is the server's private key, and y is the OPRF output. The client learns y and the server learns nothing. In the verifiable mode, the client also gets proof that the server used skS in computing the function.

To achieve verifiability, as in the original work of [JKK14], we provide a zero-knowledge proof that the key provided as input by the server in the Evaluate function is the same key as it used to produce their public key. As an example of the nature of attacks that this prevents, this ensures that the server uses the same private key for computing the VOPRF output and does not attempt to "tag" individual servers with select keys. This proof must not reveal the server's long-term private key to the client.

The following one-byte values distinguish between these two modes:

Mode	Value
modeBase	0x00
modeVerifiable	0x01

Table 1

3.1. Overview

Both participants agree on the mode and a choice of ciphersuite that is used before the protocol exchange. Once established, the base mode of the protocol runs to compute output = F(skS, input) as follows:

```
     Client(input)                                Server(skS)
  ------------------------------------------------------------
    blind, blindedElement = Blind(input)

                         blindedElement
                          ---------->

    evaluatedElement, proof = Evaluate(skS, blindedElement)

                        evaluatedElement
                          <----------

    output = Finalize(input, blind, evaluatedElement, blindedElement)
```

In Blind the client generates a token and blinding data. The server computes the (V)OPRF evaluation in Evaluation over the client's blinded token. In Finalize the client unblinds the server response and produces a byte array corresponding to the output of the OPRF protocol.

In the verifiable mode of the protocol, the server additionally computes a proof in Evaluate. The client verifies this proof using the server's expected public key before completing the protocol and producing the protocol output.

3.2. Context Setup

Both modes of the OPRF involve an offline setup phase. In this phase, both the client and server create a context used for executing the online phase of the protocol. The key pair (skS, pkS) should be generated by calling either GenerateKeyPair or DeriveKeyPair.

The base mode setup functions for creating client and server contexts are below:

```
def SetupBaseServer(suite, skS):
  contextString =
    "VOPRF07-" || I2OSP(modeBase, 1) || I2OSP(suite.ID, 2)
  return ServerContext(contextString, skS)

def SetupBaseClient(suite):
  contextString =
    "VOPRF07-" || I2OSP(modeBase, 1) || I2OSP(suite.ID, 2)
  return ClientContext(contextString)
```

The verifiable mode setup functions for creating client and server contexts are below:

```
def SetupVerifiableServer(suite, skS, pkS):
  contextString =
    "VOPRF07-" || I2OSP(modeVerifiable, 1) || I2OSP(suite.ID, 2)
  return VerifiableServerContext(contextString, skS)

def SetupVerifiableClient(suite, pkS):
  contextString =
    "VOPRF07-" || I2OSP(modeVerifiable, 1) || I2OSP(suite.ID, 2)
  return VerifiableClientContext(contextString, pkS)
```

Each setup function takes a ciphersuite from the list defined in Section 4. Each ciphersuite has a two-byte field ID used to identify the suite.

[[RFC editor: please change "VOPRF07" to "RFCXXXX", where XXXX is the final number, here and elsewhere before publication.]]

3.3. Context APIs

In this section, we detail the APIs available on the client and server (V)OPRF contexts. Each API has the following implicit parameters:

- GG, a prime-order group implementing the API described in Section 2.1.
- contextString, a domain separation tag taken from the client or server context.

The data type ClientInput is an opaque byte string of arbitrary length no larger than 2^13 octets. Proof is a concatenated sequence of two SerializedScalar values, as shown below.

```
SerializedScalar Proof[2*Ns];
```

3.3.1. Server Context

The ServerContext encapsulates the context string constructed during setup and the (V)OPRF key pair. It has three functions, Evaluate, FullEvaluate and VerifyFinalize described below. Evaluate takes serialized representations of blinded group elements from the client as inputs.

FullEvaluate takes ClientInput values, and it is useful for applications that need to compute the whole OPRF protocol on the server side only.

VerifyFinalize takes ClientInput values and their corresponding output digests from Finalize as input, and returns true if the inputs match the outputs.

Note that VerifyFinalize and FullEvaluate are not used in the main OPRF protocol. They are exposed as an API for building higher-level protocols.

3.3.1.1. Evaluate

```
Input:

  Scalar skS
  SerializedElement blindedElement

Output:

  SerializedElement evaluatedElement

Errors: DeserializeError

def Evaluate(skS, blindedElement):
  R = GG.DeserializeElement(blindedElement)
  Z = skS * R
  evaluatedElement = GG.SerializeElement(Z)

  return evaluatedElement
```

3.3.1.2. FullEvaluate

```
Input:

  Scalar skS
  ClientInput input

Output:

  opaque output[Nh]

def FullEvaluate(skS, input):
  P = GG.HashToGroup(input)
  T = skS * P
  issuedElement = GG.SerializeElement(T)

  finalizeDST = "Finalize-" || contextString
  hashInput = I2OSP(len(input), 2) || input ||
              I2OSP(len(issuedElement), 2) || issuedElement ||
              I2OSP(len(finalizeDST), 2) || finalizeDST

  return Hash(hashInput)
```

3.3.1.3. VerifyFinalize

```
Input:

  Scalar skS
  ClientInput input
  opaque output[Nh]

Output:

  boolean valid

def VerifyFinalize(skS, input, output):
  T = GG.HashToGroup(input)
  element = GG.SerializeElement(T)
  issuedElement = Evaluate(skS, [element])
  E = GG.SerializeElement(issuedElement)

  finalizeDST = "Finalize-" || contextString
  hashInput = I2OSP(len(input), 2) || input ||
              I2OSP(len(E), 2) || E ||
              I2OSP(len(finalizeDST), 2) || finalizeDST

  digest = Hash(hashInput)

  return CT_EQUAL(digest, output)
```

3.3.2. VerifiableServerContext

The VerifiableServerContext extends the base ServerContext with an augmented Evaluate() function. This function produces a proof that skS was used in computing the result. It makes use of the helper functions GenerateProof and ComputeComposites, described below.

3.3.2.1. Evaluate

```
Input:

  Scalar skS
  Element pkS
  SerializedElement blindedElement

Output:

  SerializedElement evaluatedElement
  Proof proof

Errors: DeserializeError

def Evaluate(skS, pkS, blindedElement):
  R = GG.DeserializeElement(blindedElement)
  Z = skS * R
  evaluatedElement = GG.SerializeElement(Z)

  proof = GenerateProof(skS, G, pkS, R, Z)

  return evaluatedElement, proof
```

The helper functions GenerateProof and ComputeComposites are defined below.

3.3.2.2. GenerateProof

```
Input:

  Scalar k
  Element A
  Element B
  Element C
  Element D

Output:

  Proof proof

def GenerateProof(k, A, B, C, D)
  Cs = [C]
  Ds = [D]
  a = ComputeCompositesFast(k, B, Cs, Ds)

  r = GG.RandomScalar()
  M = a[0]
  Z = a[1]
  t2 = r * A
  t3 = r * M

  Bm = GG.SerializeElement(B)
  a0 = GG.SerializeElement(M)
  a1 = GG.SerializeElement(Z)
  a2 = GG.SerializeElement(t2)
  a3 = GG.SerializeElement(t3)

  challengeDST = "Challenge-" || contextString
  h2Input = I2OSP(len(Bm), 2) || Bm ||
            I2OSP(len(a0), 2) || a0 ||
            I2OSP(len(a1), 2) || a1 ||
            I2OSP(len(a2), 2) || a2 ||
            I2OSP(len(a3), 2) || a3 ||
            I2OSP(len(challengeDST), 2) || challengeDST

  c = GG.HashToScalar(h2Input)
  s = (r - c * k) mod p
  proof = [GG.SerializeScalar(c), GG.SerializeScalar(s)]

  return proof
```

3.3.2.2.1. Batching inputs

Unlike other functions, ComputeComposites takes lists of inputs, rather than a single input. Applications can take advantage of this functionality by invoking GenerateProof on batches of inputs to

produce a combined, constant-size proof. (In the pseudocode above, the single inputs blindedElement and evaluatedElement are passed as one-item lists to ComputeComposites.)

In particular, servers can produce a single, constant-sized proof for N client inputs sent in a single request, rather than one proof per client input. This optimization benefits clients and servers since it amortizes the cost of proof generation and bandwidth across multiple requests.

3.3.2.2.2. Fresh Randomness

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 skS as is possible in Schnorr or (EC)DSA scenarios where fresh randomness is not used.

3.3.2.3. ComputeComposites

The definition of ComputeComposites is given below. This function is used both on generation and verification of the proof.

```
Input:

  Element B
  Element Cs[m]
  Element Ds[m]

Output:

  Element composites[2]

def ComputeComposites(B, Cs, Ds):
  Bm = GG.SerializeElement(B)
  seedDST = "Seed-" || contextString
  compositeDST = "Composite-" || contextString

  h1Input = I2OSP(len(Bm), 2) || Bm ||
            I2OSP(len(seedDST), 2) || seedDST
  seed = Hash(h1Input)

  M = GG.Identity()
  Z = GG.Identity()
  for i = 0 to m-1:
    Ci = GG.SerializeElement(Cs[i])
    Di = GG.SerializeElement(Ds[i])
    h2Input = I2OSP(len(seed), 2) || seed || I2OSP(i, 2) ||
              I2OSP(len(Ci), 2) || Ci ||
              I2OSP(len(Di), 2) || Di ||
              I2OSP(len(compositeDST), 2) || compositeDST
    di = GG.HashToScalar(h2Input)
    M = di * Cs[i] + M
    Z = di * Ds[i] + Z

 return [M, Z]
```

If the private key is known, as is the case for the server, this function can be optimized as shown in ComputeCompositesFast below.

```
Input:

  Scalar k
  Element B
  Element Cs[m]
  Element Ds[m]

Output:

  Element composites[2]

def ComputeCompositesFast(k, B, Cs, Ds):
  Bm = GG.SerializeElement(B)
  seedDST = "Seed-" || contextString
  compositeDST = "Composite-" || contextString

  h1Input = I2OSP(len(Bm), 2) || Bm ||
            I2OSP(len(seedDST), 2) || seedDST
  seed = Hash(h1Input)

  M = GG.Identity()
  for i = 0 to m-1:
    Ci = GG.SerializeElement(Cs[i])
    Di = GG.SerializeElement(Ds[i])
    h2Input = I2OSP(len(seed), 2) || seed || I2OSP(i, 2) ||
              I2OSP(len(Ci), 2) || Ci ||
              I2OSP(len(Di), 2) || Di ||
              I2OSP(len(compositeDST), 2) || compositeDST
    di = GG.HashToScalar(h2Input)
    M = di * Cs[i] + M

  Z = k * M

 return [M, Z]
```

3.3.3. Client Context

The ClientContext encapsulates the context string constructed during setup. It has two functions, Blind() and Finalize(), as described below. It also has an internal function, Unblind(), which is used by Finalize. The implementation of these functions varies depending on the mode.

3.3.3.1. Blind

In this mode, blinding is done multiplicatively. Under certain application circumstances, the more optimal additive blinding mechanism described in Section 3.3.4.2 can be used. See Section 6.6 for more details.

Blind is implemented as follows.

```
Input:

  ClientInput input

Output:

  Scalar blind
  SerializedElement blindedElement

def Blind(input):
  blind = GG.RandomScalar()
  P = GG.HashToGroup(input)
  blindedElement = GG.SerializeElement(blind * P)

  return blind, blindedElement
```

The inverse Unblind is implemented as follows.

```
Input:

  Scalar blind
  SerializedElement evaluatedElement

Output:

  SerializedElement unblindedElement

Errors: DeserializeError

def Unblind(blind, evaluatedElement, ...):
  Z = GG.DeserializeElement(evaluatedElement)
  N = (blind^(-1)) * Z
  unblindedElement = GG.SerializeElement(N)

  return unblindedElement
```

3.3.3.2. Finalize

Finalize depends on the internal Unblind function. In this mode, Finalize and does not include all inputs listed in Section 3.1. These additional inputs are only useful for the verifiable mode, described in Section 3.3.4.3.

```
Input:

  ClientInput input
  Scalar blind
  SerializedElement evaluatedElement

Output:

  opaque output[Nh]

def Finalize(input, blind, evaluatedElement):
  unblindedElement = Unblind(blind, evaluatedElement)

  finalizeDST = "Finalize-" || contextString
  hashInput = I2OSP(len(input), 2) || input ||
              I2OSP(len(unblindedElement), 2) || unblindedElement ||
              I2OSP(len(finalizeDST), 2) || finalizeDST
  return Hash(hashInput)
```

3.3.4. VerifiableClientContext

The VerifiableClientContext extends the base ClientContext with the desired server public key pkS with an augmented Unblind() function. This function verifies an evaluation proof using pkS. It makes use of the helper function ComputeComposites described above. It has one helper function, VerifyProof(), defined below.

3.3.4.1. VerifyProof

This algorithm outputs a boolean verified which indicates whether the proof inside of the evaluation verifies correctly, or not.

```
Input:

  Element A
  Element B
  Element C
  Element D
  Proof proof

Output:

  boolean verified

def VerifyProof(A, B, C, D, proof):
  Cs = [C]
  Ds = [D]

  a = ComputeComposites(B, Cs, Ds)
  c = GG.DeserializeScalar(proof[0])
  s = GG.DeserializeScalar(proof[1])

  M = a[0]
  Z = a[1]
  t2 = ((s * A) + (c * B))
  t3 = ((s * M) + (c * Z))

  Bm = GG.SerializeElement(B)
  a0 = GG.SerializeElement(M)
  a1 = GG.SerializeElement(Z)
  a2 = GG.SerializeElement(t2)
  a3 = GG.SerializeElement(t3)

  challengeDST = "Challenge-" || contextString
  h2Input = I2OSP(len(Bm), 2) || Bm ||
            I2OSP(len(a0), 2) || a0 ||
            I2OSP(len(a1), 2) || a1 ||
            I2OSP(len(a2), 2) || a2 ||
            I2OSP(len(a3), 2) || a3 ||
            I2OSP(len(challengeDST), 2) || challengeDST

  expectedC  = GG.HashToScalar(h2Input)

  return CT_EQUAL(expectedC, c)
```

3.3.4.2. Verifiable Blind

In this mode, where the server public key is available for proof verification, blinding is done additively. This variant is named VerifiableBlind and VerifiableUnblind. It takes two inputs: the client input and a blinded version of the group generator. The

latter is computed using a function called VerifiablePreprocess, also described below.

VerifiableBlind is implemented as follows.

```
Input:

  ClientInput input
  Element blindedGenerator

Output:

  SerializedElement blindedElement

def VerifiableBlind(input, blindedGenerator):
  P = GG.HashToGroup(input)
  blindedElement = GG.SerializeElement(P + blindedGenerator)

  return blindedElement
```

The inverse VerifiableUnblind is implemented as follows. This function can raise an exception if element deserialization or proof verification fails.

```
Input:

  Element blindedPublicKey
  SerializedElement evaluatedElement
  SerializedElement blindedElement
  Element pkS
  Scalar proof

Output:

  SerializedElement unblindedElement

Errors: DeserializeError, VerifyError

def VerifiableUnblind(blindedPublicKey, evaluatedElement,
                      blindedElement, pkS, proof):
  Z = GG.DeserializeElement(evaluatedElement)
  R = GG.DeserializeElement(blindedElement)
  if VerifyProof(G, pkS, R, Z, proof) == false:
    raise VerifyError

  N := Z - blindedPublicKey
  unblindedElement = GG.SerializeElement(N)

  return unblindedElement
```

The internal VerifiablePreprocess function computes a blind and uses it to compute a corresponding blinded version of the group generator and server public key. This function can be used to pre-compute tables of values for future invocations of the protocol, or it can be computed only when needed for a single invocation of the protocol.

```
Input:

  Element pkS

Output:

  Element blindedGenerator
  Element blindedPublicKey
  Scalar blind

def Preprocess(pkS):
  blind = GG.RandomScalar()
  blindedGenerator = ScalarBaseMult(blind)
  blindedPublicKey = pkS * blind

  return blindedGenerator, blindedPublicKey, blind
```

3.3.4.3. Verifiable Finalize

```
Input:

  ClientInput input
  Element blindedPublicKey
  SerializedElement evaluatedElement
  SerializedElement blindedElement
  Element pkS
  Scalar proof

Output:

  opaque output[Nh]

def Finalize(input, blindedPublicKey, evaluatedElement,
             blindedElement, pkS, proof):
  unblindedElement =
    VerifiableUnblind(blindedPublicKey, evaluatedElement,
                      blindedElement, pkS, proof)

  finalizeDST = "Finalize-" || contextString
  hashInput = I2OSP(len(input), 2) || input ||
              I2OSP(len(unblindedElement), 2) || unblindedElement ||
              I2OSP(len(finalizeDST), 2) || finalizeDST
  return Hash(hashInput)
```

4. Ciphersuites

A ciphersuite (also referred to as 'suite' in this document) for the protocol wraps the functionality required for the protocol to take place. This ciphersuite should be available to both the client and server, and agreement on the specific instantiation is assumed throughout. A ciphersuite contains instantiations of the following functionalities:

- GG: A prime-order group exposing the API detailed in Section 2.1, with base point defined in the corresponding reference for each group. Each group also specifies HashToGroup, HashToScalar, and serialization functionalities. For HashToGroup, the domain separation tag (DST) is constructed in accordance with the recommendations in [I-D.irtf-cfrg-hash-to-curve], Section 3.1. For HashToScalar, each group specifies an integer order that is used in reducing integer values to a member of the corresponding scalar field.

- Hash: A cryptographic hash function that is indifferentiable from a Random Oracle, whose output length is Nh bytes long.

This section specifies ciphersuites with supported groups and hash functions. For each ciphersuite, contextString is that which is computed in the Setup functions.

Applications should take caution in using ciphersuites targeting P-256 and ristretto255. See Section 6.2 for related discussion.

4.1. OPRF(ristretto255, SHA-512)

- Group: ristretto255 [RISTRETTO]

  - HashToGroup(): Use hash_to_ristretto255 [I-D.irtf-cfrg-hash-to-curve] with DST = "HashToGroup-" || contextString, and expand_message = expand_message_xmd using SHA-512.

  - HashToScalar(): Compute uniform_bytes using expand_message = expand_message_xmd, DST = "HashToScalar-" || contextString, and output length 64, interpret uniform_bytes as a 512-bit integer in little-endian order, and reduce the integer modulo Order().

  - Serialization: Both group elements and scalars are encoded in Ne = Ns = 32 bytes. For group elements, use the 'Encode' and 'Decode' functions from [RISTRETTO]. For scalars, ensure they are fully reduced modulo Order() and in little-endian order.

- Hash: SHA-512, and Nh = 64.

*ID: 0x0001

4.2. OPRF(decaf448, SHAKE-256)

*Group: decaf448 [RISTRETTO]

  - HashToGroup(): Use hash_to_decaf448 [I-D.irtf-cfrg-hash-to-curve] with DST = "HashToGroup-" || contextString, and expand_message = expand_message_xof using SHAKE-256.

  - HashToScalar(): Compute uniform_bytes using expand_message = expand_message_xof, DST = "HashToScalar-" || contextString, and output length 64, interpret uniform_bytes as a 512-bit integer in little-endian order, and reduce the integer modulo Order().

  - Serialization: Both group elements and scalars are encoded in Ne = Ns = 56 bytes. For group elements, use the 'Encode' and 'Decode' functions from [RISTRETTO]. For scalars, ensure they are fully reduced modulo Order() and in little-endian order.

*Hash: SHAKE-256, and Nh = 64.

*ID: 0x0002

4.3. OPRF(P-256, SHA-256)

*Group: P-256 (secp256r1) [x9.62]

  - HashToGroup(): Use hash_to_curve with suite P256_XMD:SHA-256_SSWU_RO_ [I-D.irtf-cfrg-hash-to-curve] and DST = "HashToGroup-" || contextString.

  - HashToScalar(): Use hash_to_field from [I-D.irtf-cfrg-hash-to-curve] using L = 48, expand_message_xmd with SHA-256, DST = "HashToScalar-" || contextString, and prime modulus equal to Order().

  - Serialization: Elements are serialized as Ne = 33 byte strings using compressed point encoding for the curve [SEC1]. Scalars are serialized as Ns = 32 byte strings by fully reducing the value modulo Order() and in big-endian order.

*Hash: SHA-256, and Nh = 32.

*ID: 0x0003

4.4. OPRF(P-384, SHA-512)

*Group: P-384 (secp384r1) [x9.62]

-HashToGroup(): Use hash_to_curve with suite P384_XMD:SHA-512_SSWU_RO_ [I-D.irtf-cfrg-hash-to-curve] and DST = "HashToGroup-" || contextString.

-HashToScalar(): Use hash_to_field from [I-D.irtf-cfrg-hash-to-curve] using L = 72, expand_message_xmd with SHA-512, DST = "HashToScalar-" || contextString, and prime modulus equal to Order().

-Serialization: Elements are serialized as Ne = 49 byte strings using compressed point encoding for the curve [SEC1]. Scalars are serialized as Ns = 48 byte strings by fully reducing the value modulo Order() and in big-endian order.

*Hash: SHA-512, and Nh = 64.

*ID: 0x0004

4.5. OPRF(P-521, SHA-512)

*Group: P-521 (secp521r1) [x9.62]

-HashToGroup(): Use hash_to_curve with suite P521_XMD:SHA-512_SSWU_RO_ [I-D.irtf-cfrg-hash-to-curve] and DST = "HashToGroup-" || contextString.

-HashToScalar(): Use hash_to_field from [I-D.irtf-cfrg-hash-to-curve] using L = 98, expand_message_xmd with SHA-512, DST = "HashToScalar-" || contextString, and prime modulus equal to Order().

-Serialization: Elements are serialized as Ne = 67 byte strings using compressed point encoding for the curve [SEC1]. Scalars are serialized as Ns = 66 byte strings by fully reducing the value modulo Order() and in big-endian order.

*Hash: SHA-512, and Nh = 64.

*ID: 0x0005

5. API Considerations

Some VOPRF APIs specified in this document are fallible. For example, Finalize and Evaluate can fail if any element received from the peer fails deserialization. The explicit errors generated

throughout this specification, along with the conditions that lead to each error, are as follows:

- *VerifyError: VOPRF proof verification failed; Section 3.3.4.2.

- *DeserializeError: Group element or scalar deserialization failure; Section 2.1.

The errors in this document are meant as a guide to implementors. They are not an exhaustive list of all the errors an implementation might emit. For example, implementations might run out of memory and return a corresponding error.

6. Security Considerations

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

6.1. 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, consider an adversary that picks inputs x from the domain of F and evaluates F on (k,x) (without knowledge of randomly sampled k). Then the output distribution F(k,x) is indistinguishable from the output distribution of a randomly chosen function with the same domain and range.

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: The server must learn nothing about the client's input or the output of the function. In addition, the client must learn nothing about the server's private key.

Essentially, obliviousness tells us that, even if the server learns the client's input x at some point in the future, then the server

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: The client must only complete execution of the protocol if it can successfully assert that the OPRF output it computes is correct. This is taken with respect to the OPRF key held by the server.

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 the server commits to the key before the actual protocol execution takes place. Then the client verifies that the server has used the key in the protocol using this commitment. In the following, we may also refer to this commitment as a public key.

6.2. Cryptographic Security

Below, we discuss the cryptographic security of the (V)OPRF protocol from Section 3, relative to the necessary cryptographic assumptions that need to be made.

6.2.1. Computational Hardness Assumptions

Each assumption states that the problems specified below are computationally difficult to solve in relation to a particular choice of security parameter sp.

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

6.2.1.1. Discrete-log (DL) Problem

Given G, a generator of GG, and H = hG for some h in GF(p); output h.

6.2.1.2. Decisional Diffie-Hellman (DDH) Problem

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

- G is a generator of GG;
- a,b are elements of GF(p);
- if d == 0: C = abG; else: C is sampled uniformly at random from GG.

Output d' == d.

6.2.2. Protocol Security

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 is identical to the [JKK14] construction, except that we can optionally perform multiple VOPRF evaluations in one go, whilst only constructing one NIZK proof object. This is enabled using an established batching technique.

Consequently, the cryptographic security of our construction is 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, k * G, and (G_1, ... , G_N), all elements of 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}, k * G_{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 6.1 under the OMDH assumption in the universal composability (UC) security model.

6.2.3. 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 GF(p)

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)/2 bits. Note that the attacks only work in situations where Q divides p-1 or p+1, where p is the order of the prime-order group used to instantiate the OPRF.

As an example, suppose that a group instantiation is used that provides 128 bits of security against discrete log cryptanalysis. 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)/2 = 10 bits. Launching an attack would require 2^(p/2-log_2(Q)/2) bits of memory.

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, k * G, (k^2)G, ..., (k^(Q-1))G), where each query is the output of the previous interaction. This means that any client that submits Q queries to the OPRF can use the aforementioned attacks to reduce the security of the group instantiation by (log_2(Q)/2) 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.

6.2.4. 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)/2) bits of security. Moreover, such attacks are only possible for those certain applications where the adversary can query the OPRF directly. In applications where such an oracle is not made available this security loss does not apply.

In most cases, 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. For applications that admit the aforementioned oracle functionality, and that cannot tolerate discrete logarithm security of lower than 128 bits, we recommend only implementing ciphersuites with IDs 0x0002, 0x0004, and 0x0005.

6.3. Domain Separation

Applications SHOULD construct input to the protocol to provide domain separation. Any system which has multiple (V)OPRF

applications should distinguish client inputs to ensure the OPRF results are separate. Guidance for constructing info can be found in [I-D.irtf-cfrg-hash-to-curve]; Section 3.1.

6.4. Element and Scalar Validation

The DeserializeElement function recovers a group element from an arbitrary byte array. This function validates that the element is a proper member of the group and is not the identity element, and returns an error if either condition is not met.

For P-256, P-384, and P-521 ciphersuites, this function performs partial public-key validation as defined in Section 5.6.2.3.4 of [keyagreement]. This includes checking that the coordinates are in the correct range, that the point is on the curve, and that the point is not the point at infinity. If these checks fail, deserialization returns an error.

For ristretto255 and decaf448, elements are deserialized by invoking the Decode function from [RISTRETTO], Section 4.3.1 and [RISTRETTO], Section 5.3.1, respectively, which returns false if the element is invalid. If this function returns false, deserialization returns an error.

The DeserializeScalar function recovers a scalar field element from an arbitrary byte array. Like DeserializeElement, this function validates that the element is a member of the scalar field and returns an error if this condition is not met.

For P-256, P-384, and P-521 ciphersuites, this function ensures that the input, when treated as a big-endian integer, is a value between 0 and Order(). For ristretto255 and decaf448, this function ensures that the input, when treated as a little-endian integer, is a valud between 0 and Order().

6.5. Hashing to Group

A critical requirement of implementing the prime-order group using elliptic curves is a method to instantiate the function GG.HashToGroup, that maps inputs to group elements. In the elliptic curve setting, this deterministically maps inputs x (as byte arrays) to uniformly chosen points on the curve.

In the security proof of the construction Hash is modeled as a random oracle. This implies that any instantiation of GG.HashToGroup must be pre-image and collision resistant. In Section 4 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.

6.6. Blinding Considerations

This document makes use of two types of blinding variants: multiplicative and additive. The advantage of additive blinding is that it allows the client to pre-process tables of blinded scalar multiplications for the group generator and server public key. This can provide a computational efficiency advantage (due to the fact that a fixed-base multiplication can be calculated faster than a variable-base multiplication). Pre-processing also reduces the amount of computation that needs to be done in the online exchange.

However, the choice of blinding mechanism has security implications. [JKX21] analyze the security properties of both blinding mechanisms used in this document. The results can be summarized as follows:

- Multiplicative blinding is safe for all applications.

- Additive blinding is possibly unsafe, unless one of the following conditions are met:

  - The client has a certified copy of the server public key (as is the case in the verifiable mode);

  - The client input has high entropy; and

  - The client mixes the public key into the OPRF evaluation.

To avoid security issues with the base mode, where some of the above conditions may not be met, this specification RECOMMENDS use of multiplicative blinding. This is because it is not known if the server public key is available or if the client input has high entropy. Applications wherein either of these conditions are true MAY use additive blinding.

The verifiable mode always makes use of the more efficient additive blinding variant, as the public key is always available for verifying the proof.

6.7. Timing Leaks

To ensure no information is leaked during protocol execution, all operations that use secret data MUST run in constant time. Operations that SHOULD run in constant time include all prime-order group operations and proof-specific operations (GenerateProof() and VerifyProof()).

6.8. 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 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 6.2.3.

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.

6.8.1. Parameter Commitments

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

7. Acknowledgements

This document resulted from the work of the Privacy Pass team [PrivacyPass]. The authors would also like to acknowledge helpful conversations with Hugo Krawczyk. Eli-Shaoul Khedouri provided additional review and comments on key consistency. Daniel Bourdrez, Tatiana Bradley, Sofia Celi, Frank Denis, and Bas Westerbaan also provided helpful input and contributions to the document.

8. References

8.1. 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>.

[Cheon06] "Security Analysis of the Strong Diffie-Hellman Problem", <https://www.iacr.org/archive/eurocrypt2006/40040001/40040001.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.davidson-pp-protocol]
Davidson, A., "Privacy Pass: The Protocol", Work in Progress, Internet-Draft, draft-davidson-pp-protocol-01, 13 July 2020, <https://datatracker.ietf.org/doc/html/draft-davidson-pp-protocol-01>.

[I-D.irtf-cfrg-hash-to-curve] Faz-Hernandez, A., Scott, S., Sullivan, N., Wahby, R. S., and C. A. Wood, "Hashing to Elliptic Curves", Work in Progress, Internet-Draft, draft-irtf-cfrg-hash-to-curve-11, 13 April 2021, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hash-to-curve-11>.

[I-D.irtf-cfrg-opaque] Krawczyk, H., Bourdrez, D., Lewi, K., and C. A. Wood, "The OPAQUE Asymmetric PAKE Protocol", Work in Progress, Internet-Draft, draft-irtf-cfrg-opaque-05, 7 June 2021, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-opaque-05>.

[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>.

[PrivacyPass] "Privacy Pass", <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/rfc/rfc2119>.

[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/rfc/rfc7748>.

[RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <https://www.rfc-editor.org/rfc/rfc8017>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RISTRETTO] Valence, H. D., Grigg, J., Tankersley, G., Valsorda, F., Lovecruft, I., and M. Hamburg, "The ristretto255 and

decaf448 Groups", Work in Progress, Internet-Draft, draft-irtf-cfrg-ristretto255-decaf448-00, 5 October 2020, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-ristretto255-decaf448-00>.

[SEC1] Standards for Efficient Cryptography Group (SECG), ., "SEC 1: Elliptic Curve Cryptography", <https://www.secg.org/sec1-v2.pdf>.

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

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

[x9.62] ANSI, "Public Key Cryptography for the Financial Services Industry: the Elliptic Curve Digital Signature Algorithm (ECDSA)", ANSI X9.62-1998, September 1998.

8.2. Informative References

[JKX21] Jarecki, S., Krawczyk, H., and J. Xu, "On the (In)Security of the Diffie-Hellman Oblivious PRF with Multiplicative Blinding", PKC'21 , March 2021, <https://eprint.iacr.org/2021/273>.

[keyagreement] Barker, E., Chen, L., Roginsky, A., Vassilev, A., and R. Davis, "Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography", National Institute of Standards and Technology report, DOI 10.6028/nist.sp.800-56ar3, April 2018, <https://doi.org/10.6028/nist.sp.800-56ar3>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/rfc/rfc8446>.

Appendix A. Test Vectors

This section includes test vectors for the (V)OPRF protocol specified in this document. For each ciphersuite specified in Section 4, there is a set of test vectors for the protocol when run in the base mode and verifiable mode. Each test vector lists the batch size for the evaluation. Each test vector value is encoded as a hexadecimal byte string. The label for each test vector value is described below.

*"Input": The client input, an opaque byte string.

*"Blind": The blind value output by Blind(), a serialized Scalar of Ns bytes long.

*"BlindedElement": The blinded value output by Blind(), a serialized Element of Ne bytes long.

*"EvaluatedElement": The evaluated element output by Evaluate(), a serialized Element of Ne bytes long.

*"EvaluationProofC": The "c" component of the Evaluation proof (only listed for verifiable mode test vectors), a serialized Scalar of Ns bytes long.

*"EvaluationProofS": The "s" component of the Evaluation proof (only listed for verifiable mode test vectors), a serialized Scalar of Ns bytes long.

*"Output": The OPRF output, a byte string of length Nh bytes.

Test vectors with batch size B > 1 have inputs separated by a comma ",". Applicable test vectors will have B different values for the "Input", "Blind", "BlindedElement", "EvaluationElement", and "Output" fields.

Base mode uses multiplicative blinding while verifiable mode uses additive blinding, as described in Section 3.3.3 and Section 3.3.4, respectively.

The server key material, pkSm and skSm, are listed under the mode for each ciphersuite. Both pkSm and skSm are the serialized values of pkS and skS, respectively, as used in the protocol. Each key pair is derived from a seed, which is listed as well, using the following implementation of DeriveKeyPair:

```
def DeriveKeyPair(mode, suite, seed):
  skS = GG.HashToScalar(seed, DST = "HashToScalar-" || contextString)
  pkS = ScalarBaseMult(skS)
  return skS, pkS
```

A.1. OPRF(ristretto255, SHA-512)

A.1.1. Base Mode

```
seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3
skSm = caeff69352df4905a9121a4997704ca8cee1524a110819eb87deba1a39ec1
701
```

A.1.1.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = c604c785ada70d77a5256ae21767de8c3304115237d262134f5e46e512cf
8e03
BlindedElement = fc20e03aff3a9de9b37e8d35886ade11ec7d85c2a1fb5bb0b16
86c64e07ac467
EvaluationElement = 7c72cc293cd7d44c0b57c273f27befd598b132edc665694b
dc9c42a4d3083c0a
Output = e3a209dce2d3ea3d84fcddb282818caebb756a341e08a310d9904314f53
92085d13c3f76339d745db0f46974a6049c3ea9546305af55d37760b2136d9b3f013
4
```

A.1.1.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 5ed895206bfc53316d307b23e46ecc6623afb3086da74189a416012be037
e50b
BlindedElement = 483d4f39de5ff77fa0f9a0ad2334dd5bf87f2cda868539d21de
67ce49e7d1536
EvaluationElement = 026f2758fc62f02a7ff95f35ec6f20186aa57c0274361655
543ea235d7b2aa34
Output = 2c17dc3e9398dadb44bb2d3360c446302e99f1fe0ec40f0b1ad25c9cf00
2be1e4b41b4900ef056537fe8c14532ccea4d796f5feab9541af48057d83c0db86fe
9
```

A.1.2. Verifiable Mode

```
seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3
skSm = ac37d5850510299406ea8eb8fa226a7bfc2467a4b070d6c7bf667948b9600
b00
pkSm = 0c0254e22063cae3e1bae02fb6fa20882664a117c0278eda6bda3372c0dd9
860
```

A.1.2.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = ed8366feb6b1d05d1f46acb727061e43aadfafe9c10e5a64e7518d63e326
3503
BlindedElement = 66a2835d9da73ecf1ea433bca7ae2b887995cf4b1ffc14f77de
0a417e880a534
EvaluationElement = 8ac09f2a720b3ff03373776edaff052ac4cbe685737df20b
82f6ae1829b21f25
EvaluationProofC = a82634a5381954a31a2f537f76a6765ed6965ed5a48599b41
064d213b06e4b0f
EvaluationProofS = 4b8e4ce683fdc433965db51efb9573a669e83ed5982b05a48
3b71fa181179e0c
Output = e694146179b874ff64f0c9ec59c292031e7d3386703d0ed129cfee98eec
eeb184c0335c7c28be75f7298a33a48c05b2acbd36169a075e495f710c490b404ce3
0
```

A.1.2.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = e6d0f1d89ad552e383d6c6f4e8598cc3037d6e274d22da3089e7afbd4171
ea02
BlindedElement = 122d4446e76662865ff48a7f1117eab928cfdaafca237e3b424
7cb9c6761fd0f
EvaluationElement = 6438480e8cbff8afbcd23f8655910388a9251874b008e1aa
9ff36f3f89c0e02a
EvaluationProofC = c7a18f8d95aceb7f5948cb3c8747b3d33900a6f3d7ccb9236
5860da919ebd705
EvaluationProofS = 67e02e92883fa976ca171b87807450af9ac638581311bd5bc
cc9efa8ddae020a
Output = aa5475f7d60613eab56bf317ea10340ed26be9cbea9685f2617f5a68d55
09a485a15d4db1b41c3ef0f68dd54cc3a4a53d92813797babb0683b71263f5e63b3f
6
```

A.1.2.3. Test Vector 3, Batch Size 2

```
Input = 00,5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 80513e77795feeec6d2c450589b0e1b178febd5c193a9fcba0d27f0a06e0
d50f,533c2e6d91c934f919ac218973be55ba0d7b234160a0d4cf3bddafbda99e2e0
c
BlindedElement = 025529520d154b88dde799ad81b4bad43c10d2d983c866304b3
65de3062d5c16,68b7ead5133c421ac2052e384421b7d306fe698245e0bebe4d0058
bb7daf1b69
EvaluationElement = e27c872c0f0e04aa86927c28b42fb3181c312741296d6dc6
247b3152ea7bb81f,f07381baa4420ed5cf1bb4e52f3e1a32a9372b17161e11086d1
18b424f87a104
EvaluationProofC = fa1af76c86aaa73a4abb577f5a30ff9b8aced055120ae0390
b73f51d173c620a
EvaluationProofS = 51ccd568a3494f627eee67660a44133706d67d88a9652aee1
02f999553214e06
Output = e694146179b874ff64f0c9ec59c292031e7d3386703d0ed129cfee98eec
eeb184c0335c7c28be75f7298a33a48c05b2acbd36169a075e495f710c490b404ce3
0,aa5475f7d60613eab56bf317ea10340ed26be9cbea9685f2617f5a68d5509a485a
15d4db1b41c3ef0f68dd54cc3a4a53d92813797babb0683b71263f5e63b3f6
```

A.2. OPRF(decaf448, SHAKE-256)

A.2.1. Base Mode

```
seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3
skSm = 78f2622804104209f7e015370ff98f4a3cbf311e6784e9f4944f8a252dc08
e916d9ab1a60dc905f0e56631903ecd4ae6e15291776d61460b
```

A.2.1.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = d1080372f0fcf8c5eace50914e7127f576725f215cc7c111673c635ce668
bbbb9b50601ad89b358ab8c23ed0b6c9d040365ec9d060868714
BlindedElement = 984e0a443ee194090737df4afb402253f216b77650c91d252b6
638e1179723d51a4154b88eae396f1320f5df3c4b17f779516c456e364bd1
EvaluationElement = ec07794d109d5dd976bac4f23a4d041f4abd17017af78add
5dd87c66db44590131e23dd3e30310f43284b9e0b874041105eb8201ad584830
Output = e2d48872c64bd0e62cd6165c0a75399bbf6cecf39b85230c2748ccf7ebf
376d28394ac0b9518e1adb070c6952b5ab897e1cbce5f9d78ec8935de59f58210321
a
```

A.2.1.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = aed1ffa44fd8f0ed16373606a3cf7df589cca86d8ea1abbf5768771dbef3
d401c74ae55ba1e28b9565e1e4018eb261a14134a4ce60c1c718
BlindedElement = 4aa751f84b2634b73efa364b03e60b92b84f457576e6b369eea
b76140e3859d10d2e98174f13f5a2c70670529ccf093d5f1aaf355b4f830b
EvaluationElement = 429e9d382354f8079cc1e353ef431243cdfca5ec35eb662f
2214b96483e5e6392f15734dd56d65c0c1600d7fd685315831d11f3c33e7230d
Output = b57a408fa6dafe00091e6f6b3d0086bc21ae9c4ead173987672e92f7116
2fea3970341cd63bc33ca442302474bbf95db41cd2b9fa0f2a607b6b4f57443a2f4b
5
```

A.2.2. Verifiable Mode

```
seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3
skSm = 9eb722f7fee9f61f24ad31bc42309f73648cf4393929e8f5f333fe10c6975
c827a1eba4e03ae2fa8735db2f63f6c98c7af6010e64c81f535
pkSm = b6e2751176d57836fe1dfbdbbdc78a1b5c5a52f831226c9d8dfdf5daf8f46
6e310e80978e9b81c387f5bc85cc7ef5567f4dd3ba7674579a2
```

A.2.2.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = 4c936db1779a621b6c71475ac3111fd5703a59b713929f36dfd1e892a7fe
814479c93d8b4b6e11d1f6fe5351e51457b665fa7b76074e531f
BlindedElement = c0c9b0d321c4e7c8c13b910428196af3f36d5dae4796e2577fb
51493d3204cc1ee7732d81e0429195e4ad9226b84797feb261bfb2dbe48e9
EvaluationElement = a8f7614c46fb27a49f22fd3d3795d4f96667dfa075d967d7
3e32e9eb0e6705055dfc4446b20c8600cb1a6e636b9066217690c9ee9cd509c0
EvaluationProofC = 199a534bc888fb31dc08a1a1374c669ae61578ba7d51c0d40
f5169e35f95eda58fa7801d30593206b4df72de0680956dff2d8ded5b56e324
EvaluationProofS = 91370471e63326a97f0b2830c4caf95129ac603fd504496fc
cb559c9109e079c49a504acbef4dd6e64ef2f429286f40ff00f4c6a7a57a107
Output = 31c251b66b1fafa46519252c9fc018dd117f219697f45e38bda7253d9cd
02d5a631600387c4863139ea79b3c036c706e70040d2148c43fa2fd47ea3d1d6f42e
5
```

A.2.2.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 15b3355179392f40c3d5a15f0d5ffc354e340454ec779f575e4573a3886a
b5e57e4da2985cea9e32f6d95539ce2c7189e1bd7462e8c5483a
BlindedElement = 3c05e826c9d885fd29bf537acc3bcc279c0821bd65cf21d9dcc
99a15af2474df8b3903f220cce9a9f7966a3991c37ae69a73deca4925ab58
EvaluationElement = ac1821113a6748a66c75ac71bedf2784402b60cbf9df16c3
cfc0ba1a240edfaa5ea5f35a5d5fa8221d60deeceb9dd14a6c4ed0ef01f2f2b4
EvaluationProofC = 2129b555a89ee00a826ae0080a71874a2b7612dfa3357c32f
3c1421cb0f93168eee21d72e0d279e772fed4724397463c4972ec8b09f50a2f
EvaluationProofS = a1ec1562977cb52a458e26736e350ac02e09f709b81788f6b
944022c2313a1217134b9cc6232ac86d6b758d2b906ac5d9158a13e7221ec32
Output = e7956cc9410d6a906a0293172e816750cfc099d04db7af8a8970b3bb20d
73cbec56d61a969ff1db724b1701cbf63ad570b591a1fde2c7f409dee29457a6c5d8
c
```

A.2.2.3. Test Vector 3, Batch Size 2

```
Input = 00,5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 614bb578f29cc677ea9e7aea3e4839413997e020f9377b63c13584156a09
a46dd2a425c41eac0e313a47e99d05df72c6e1d58e6592577a0d,4c115060bca87db
7d73e00cbb8559f84cb7a221b235b0950a0ab553f03f10e1386abe954011b7da62bb
6599418ef90b5d4ea98cc28aff517
BlindedElement = e2f4aa0e9d09c39cc6f9f58fa4e99af814977f3033cfe613d7c
e525ebf62b93970e4e32b99c1f8090b55d55fb9cb6180c864b5ea29342f98,6ec406
36ebae90b96df53d451c47a26231c7975ab0b3f2af61f8b6d6455662a08cdeb92ccb
6bc08d6c0a6662435f693f55c043bd4d366b01
EvaluationElement = 32f832dcf7d7d493c1c23c3cd94546aef8ffbc455ac16ebe
c2d892d9e731581b41790879e735cb825574552af397a787b47593f095f5a40f,ea5
eee6f8206ceaa109b2c0fcf7807d3b585c751e1476ab140515972eac3a9a847b4858
871e4d92c69a7c8e32bcbaa2aa59d876f02b083b8
EvaluationProofC = 170d265464be75ee7e820f4b90b4f6ded3c379607b103eeb9
e04026f07eb89ae233169af78b32d79dfa40a7bbce0d16c6a3562345e44471f
EvaluationProofS = ec007985eb27fa7931793801f3e060b272ac12610419ca411
52b24e7fab94e639cd85d95b9f6fa16ec32715fd4028f6441d4c5e06a0e2c3c
Output = 31c251b66b1fafa46519252c9fc018dd117f219697f45e38bda7253d9cd
02d5a631600387c4863139ea79b3c036c706e70040d2148c43fa2fd47ea3d1d6f42e
5,e7956cc9410d6a906a0293172e816750cfc099d04db7af8a8970b3bb20d73cbec5
6d61a969ff1db724b1701cbf63ad570b591a1fde2c7f409dee29457a6c5d8c
```

A.3. OPRF(P-256, SHA-256)

A.3.1. Base Mode

```
seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3
skSm = a1b2355828f2c76de6749af9d093bd9fe0f2cada3ec653cd9a6d3126a7a78
27b
```

A.3.1.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = 5d9e7f6efd3093c32ecceabd57fb03cf760c926d2a7bfa265babf29ec98a
f0d0
BlindedElement = 03e3c379698da853d9844098fa0ac676970d5ec24167b598714
cd2ee188604ddd2
EvaluationElement = 03ea54e8d095332d1a601a3f8a5013188aea036bf9b56323
6f7fd3b046908b42fd
Output = 464e3e51e4086a824d9a2f939524d7069ae4072a788bc9d5daa0762b258
26437
```

A.3.1.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 825155ab61f17605af2ae2e935c78d857c9407bcd45128d57d338f1671b5
fcbe
BlindedElement = 030b40be181ffbb3c3ae4a4911287c43261f5e4034781def69c
51608f372a02102
EvaluationElement = 03115ad70ea55dbb4006da0ee3589a3582f31ef9cd143996
d1e31a25ad3abdcf6f
Output = b597d58c843d0f9d2712121b0a3e2912ebee1c829eed3089eade9af4359
ab275
```

A.3.2. Verifiable Mode

```
seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3
skSm = 4e7804245a743c59d624457677294e04a8bc4bdcd94f0d3bd54f568067489
d34
pkSm = 03b51a0af95c819b09ee80c2056cf0ab0551a5355266d3a0aaff90c3fe915
ed892
```

A.3.2.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = cee64d86fd20ab4caa264a26c0e3d42fb773b3173ba76f9588c9b14779bd
8d91
BlindedElement = 03a734663e3eb29474c79996fe1d0662071594ed7a27832e214
9bad364e44784c5
EvaluationElement = 022ca51b981dd6bd5e08ad7f16ed0e08e67270fe2fc0df76
0be8f95019ffba50c6
EvaluationProofC = d7403f40351dc210d55b78ee34671b2f239faf264b5af8aec
a3101ada4ff8643
EvaluationProofS = 6d7cb0e2697bf197578e6dde7abf09d31cf782260e471c930
e79acc6949df83e
Output = 63e261480afa379117f23016997d6c8558262407d54c588881b74650a7c
cdfd1
```

A.3.2.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 5c4b401063eff0bf242b4cd534a79bacfc2e715b2db1e7a3ad4ff8af1b24
daa2
BlindedElement = 03b6bc0a56d6ed8461a06605c1b293e768d418dc9c42b8851c1
a49292456dd1dad
EvaluationElement = 0349529dc853a564345aa04a9512a088ad69c5064b1108f8
edfe914c5b74e03f58
EvaluationProofC = 88185c7c3a5fdebf24fc37c39925d745b4a286ec9234f32a3
dbf36c5b5261230
EvaluationProofS = aac945f48e297e4d70ed819688176030ffe370bfe2292a12c
4570b83ba7912a0
Output = 1232618e21b6c287ac80fcbe3fbc5013d6ec858a1529abf628514c0c19c
16570
```

A.3.2.3. Test Vector 3, Batch Size 2

```
Input = 00,5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = f0c7822ba317fb5e86028c44b92bd3aedcf6744d388ca013ef33edd36930
4eda,3b9631be9f8b274d9aaf671bfb6a775229bf435021b89c683259773bc686956
b
BlindedElement = 02fbfb72be2e5df24abb1791489e3a77f9d30b28bafea7ba7c0
f2c31bbf3deb851,0357f017a6cb76f545233c96b806165b52fb7ab700708f817cf3
501b789f8d57fb
EvaluationElement = 0350f7984730eb0a6fa95e795b68a13b3d461bf0c9bfa6e4
a33709b08f65ae528b,026061aa05c776aa08fd96c7eb284051f8a1303d989a1c8b5
e9a478e5c9f37207f
EvaluationProofC = a6e08a019deaf2eba7f40162be527bb574db98679662bb97e
0db02d58638d0dc
EvaluationProofS = 65dc2e6b64e172d8b7f6dac088737f4612a1ea455801eed6d
140554bed0903dd
Output = 63e261480afa379117f23016997d6c8558262407d54c588881b74650a7c
cdfd1,1232618e21b6c287ac80fcbe3fbc5013d6ec858a1529abf628514c0c19c165
70
```

A.4. OPRF(P-384, SHA-512)

A.4.1. Base Mode

```
seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3
skSm = ef1b52c12cdf43dc260bf5425a30cde7d708ec34b38dcfbdc2946d7baf525
361e797f6a98f1ebd80f64865f21cde1c6d
```

A.4.1.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = 359073c015b92d15450f7fb395bf52c6ea98384c491fe4e4d423b59de7b0
df382902c13bdc9993d3717bda68fc080b99
BlindedElement = 02fa3115c21ffcacc09ca470729b725781f84333e217cfeec2b
8ba6a54ce492ede7ead3714c5b177427ef853effb1b5c24
EvaluationElement = 020407d59b1497e23f6ff1af68631657dd31c418a8f5e3e9
4878e842ace8b72514f90f95909284069ca88b7728ba5fcdc8
Output = a1a1e468268164ad307b7bb5a92add28ed42117f5ca4dfd6e6534c56183
4ac65f394d0eb25489a22420d52c605931745a6683638b648d230b7599c7c78cb0b1
5
```

A.4.1.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 21ece4f9b6ffd01ce82082545413bd9bb5e8f3c63b86ae88d9ce0530b01c
b1c23382c7ec9bdd6e75898e4877d8e2bc17
BlindedElement = 025fddc89a832089a59120df742acb34dba82b26afcae977961
57df238b5905c494a23c56b1f485cbbff78d31df7fa1492
EvaluationElement = 03ab50834f953a0b27afdad59cb18c6ffa3d95dfc265467b
d7dfe96935b6cef735daa39f7a08853092f47323371695153d
Output = b5ef8dbe81d0fd8bbcd104d5f729aaa1d8b7378ab0c6da84c1a13152891
ac71a069bc9654f7f6c3f9dfc6b9e5d47996e81d7679992300216afe9a759f737741
e
```

A.4.2. Verifiable Mode

```
seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3
skSm = 67ee1c9e67566d87bfcca9e5dac4bfdb8bdd727c031133fac2aa9ba6c41e6
1e5f8fd401b5d76c7d54b15b15932797479
pkSm = 029b51b2ce9c499f2056e65e0f41d60960f9c4795c0cf94af273ce840c20b
e4cdf87690b6b121b37d399b49afcc2ec9ac3
```

A.4.2.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = 102f6338df84c9602bfa9e7d690b1f7a173d07e6d54a419db4a6308f8b09
589e4283efb9cd1ee4061c6bf884e60a8774
BlindedElement = 03dacf29b7369164ab41eb0388a8258def9ff09933b3a2abed4
a400327810b0b5d940a4d4710473f61266a6d2b20e6786d
EvaluationElement = 020b0ee924fc9a6f9b51894b3db9df3f2e1f29eb46f41a59
7aaa81942e1f6673f38c47db05924d08867cc7b45ffd5bfba0
EvaluationProofC = 79fe0dbfc6fd433b155f98fd4a019de4d7ffeef4bd6eb10a8
bf5a03602f20ef99ac2e05ca497f6276338b6f523bcf4b2
EvaluationProofS = 4ad2a121801039a9368f3aeb7011b553d18e7f4406c63ddfb
5799a9f53e3fc11c939b591ebc670a7297c66624d62ff49
Output = 8cf810d1e5750dffe92ad14749c007bbcfea47153eb49007edafd30201d
deca63730db8acdec0ca7fd8db2a96775e3f17c7d41db40d469c1054ac0d3f709cf2
b
```

A.4.2.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 8aec1d0c3d16afd032da7ba961449a56cec6fb918e932b06d5778ac7f67b
ecfb3e3869237f74106241777f230582e84a
BlindedElement = 027c6fc6f8ddf937b6c44183a39246ab26d0caf5ad7dd13e556
acbb997cd66295e66511a78df5e216a4d354bc7e909e027
EvaluationElement = 025b559f59b4e3417bf26956af2675c9e3f3cd5d97e1c507
8dc880e8bfb0858a34fb9ba9744b7d9d66effc3e8faf02371a
EvaluationProofC = 144948b5d60dbf8b3cb15a5808e6cd35715db57e0c5e4e237
1a0d7b18bdea59e9e0e53dd9e618010544c5e33a52c6565
EvaluationProofS = f237c5f04902be869bd21b94118c58fe2d1aacb182bab2271
ea107afc9da3a97aadaac27022b46980ffd5ab828c54df5
Output = 2c7f0e808bd75781c1673bfa14de23347a0ef3a4c413312bc4368387ae3
bd6da250f726f1b5dcade0187f0ac378e54fdd3792e3a66ce9482c42631909e9de1d
8
```

A.4.2.3. Test Vector 3, Batch Size 2

```
Input = 00,5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 41fabd4722d92472d858051ce9ad1a533176a862c697b2c392aff2aeb77e
b20c2ae6ba52fe31e13e03bf1d9f39878b23,51171628f1d28bb7402ca4aea6465e2
67b7f977a1fb71593281099ef2625644aee0b6c5f5e6e01a2b052b3bd4caf539b
BlindedElement = 030117571255e98fe43636c07b1d6a2e793563bf506449461447
295678c707c4e9214d9805fcc3df35acfb29b775fab1f0,039e9c4a303efc469e20
00965c7a023dabc43f88caea4e8b47e3b737ee8fc30ba93480d29b9e1e3e59cedfa2
a5f4aeb9dd
EvaluationElement = 032a382fdef3f63c6bd4fbdc26448773356880a1d7e16ec4
0f35374714e49cc725ea84e33b69e188f68fb782c558263ce0,02f88eaba928531d8
357e808cb7148c0de204d5d589a6bba770c4cfa448876a029aea214c3e0bb6a693a7
d88b001dbc098
EvaluationProofC = 0bc83100bf6b3ac8d8ad46f3973c107441edfa551f5622286
751698617f809713e749492e1310241bd0f66c657d74f8b
EvaluationProofS = 09b2fc47495de88f2dbde5ea6f0abc24d750861b6a25adca3
a36e65ac1e0913a2f988e0684e520fd3b63be47e57914b4
Output = 8cf810d1e5750dffe92ad14749c007bbcfea47153eb49007edafd30201d
deca63730db8acdec0ca7fd8db2a96775e3f17c7d41db40d469c1054ac0d3f709cf2
b,2c7f0e808bd75781c1673bfa14de23347a0ef3a4c413312bc4368387ae3bd6da25
0f726f1b5dcade0187f0ac378e54fdd3792e3a66ce9482c42631909e9de1d8
```

A.5. OPRF(P-521, SHA-512)

A.5.1. Base Mode

```
seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3
3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3
skSm = 016ee706f30ce7e15e4ffa3114c7d59a7b6f302d531ca60419be39d1cd43e
e13b1fc8398b7f63a900cdc49c6e99f65a74403db2fa739927a2ee288cff857d9d84
ecf
```

A.5.1.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = 01b983705fcc9a39607288b935b0797ac6b3c4b2e848823ac9ae16b3a3b5
816be03432370deb7c3c17d9fc7cb4e0ce646e04e42d638e0fa7a434ed340772a8b5
d626
BlindedElement = 0301f0a8c68e58f5571bd39fe3b0b2aa055a8c34e3d68ba0d2e
d177db0bc7575d477ed8f557596feb5ac568fe738eee8cff7dcb56dc78f52bf381c0
912e0e84b5a3f5b
EvaluationElement = 0200babff80ca9060825acc2e53da599016aa4685f3c3f38
ad71dafe4200b26a588cd040fb56947c78929fc1bc56075bedbb363f28bcf65e33c6
fe7f938783a2dfa97e
Output = da67996ac63b34db17ae15c98edd21565286d7549970f260c2f1360c62e
418c3695fa61e50c39df59be2e14376abb6732c07a627a7b49ac4615544e95b47afa
c
```

A.5.1.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 01a03b1096b0316bc8567c89bd70267d35c8ddcb2be2cdc867089a2eb5cf
471b1e6eb4b043b9644c8539857abe3a2022e9c9fd6a1695bbabe8add48bcd149ff3
b841
BlindedElement = 030099c35342a43221c6e03debfb17bad71b62e04c9242aa6e9
f2f915163ef4f5b8b7fe1740a4d636c36bd5c73ca39c69992dc7f6dff8f232125efc
22af4df8352fea2
EvaluationElement = 03017c756b73ddbaa80c2ee4ad61e431ac2494f80c258065
23c6be5cff2f544540d74263a4c93fb4e43e8411fc145448013c376d7ddb73f001cb
c9c5529b693b0e1aa8
Output = 3d8405a1f73a8c2fb87bb1136499991fe0c4708b64767531385dc6db106
f09d47868860fbb47bc1ac6a466673bb52ef304cba124580109ea41b2734eb3cd14c
5
```

A.5.2. Verifiable Mode

```
seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3
skSm = 0017674057e06c5e3e8a331f2dc3558540701c9cd0f4c19126d5972af6a01
447b312d05a06dab3e9e07c891d749444c27ede0897ad42aea03b887eb5db93e3f29
a86
pkSm = 0201ee4e2eaa74728f577f4bb282c5440cd454fdee1d79b15a36d34b5e5a1
25e3ccc0f99e32cc0a6a15b5652a0c8a424860c6753f685d0e1e150ceba24ca3386f
29216
```

A.5.2.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = 00bbb82117c88bbd91b8954e16c0b9ceed3ce992b198be1ebfba9ba970db
d75beefbfc6d056b7f7ba1ef79f4facbf2d912c26ce2ecc5bb8d66419b379952e96b
d6f5
BlindedElement = 03017f6448132acb88e7b3c4324f9d9608e27a95dc9ac643915
a9e85618ed09b7072729237646da9a0d37a1169c148cb13041817ccf6e2eb37b512f
f89d1850718ec50
EvaluationElement = 0300db733bf9d6441b04d9f650b3a572252200f84dc15d67
e75d6417479db3dc5c1f29cefc0e3578858ad3ec6037d3ea2f03b2d60ba5fe9f6aa4
b408ee128204bbb334
EvaluationProofC = 002097435da1befb7ad15ad2d25ad3e56b0eccbec5087d773
94039c2871a020dcf30deb4d95b5050f5febe1ebda4547870d46b1d3cef4dd04f793
e6016a23d180285
EvaluationProofS = 01976fbe0ce1922c14eef7ad402cb463d6f11ba2ae402a19e
4bc514c814b42d226175f76353c2b0bad26f98985e5f3b96a8e5557d78b7bc13eb8c
44cafb9f441772c
Output = 64b90ccb524dc125290bda4f6102871a8580d9cb0dfe1c0e46ae87d70a1
690ddf7ed5d970ac3b51e959a8769fd8e5ee5a92ebf4e74db6104fc4334258b128d5
0
```

A.5.2.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 009055c99bf9591cb0eab2a72d044c05ca2cc2ef9b609a38546f74b6d688
f70cf205f782fa11a0d61b2f5a8a2a1143368327f3077c68a1545e9aafbba6a90dc0
d40a
BlindedElement = 0201a8d0747e840cedd4586ca5382ef51031c9fb7074fcf6c9b
7acd6c74ee80dd33efa01f91cd3f50c9685fdea84c8a779cf8a3a6caedeb8050a782
241f8469d641d1a
EvaluationElement = 0200470a42df4d40ddaebb7e4a3e308c09396e4f0217fb62
ffd1aad6f35449c41e655a0b073a82003b192685dc5567096a2ad690128dfdd62693
c97cd54e536b044ad5
EvaluationProofC = 01c01a155fd0c0e690f44cd2e4771b9c4cbc95d84e517f103
79b5e59e643c4c28b480b2a582e8dc03f1fb1f9dd26b73a7fbc9443c5cf543d288d4
3d078fe9db1f3a1
EvaluationProofS = 01bb2e45130b23cf6af1ac7777ac1c6189d0b3740c6fb1130
69bed943fd19d4093c6c1ffd872964189a02f5948d7f06d2090321510a46beabe6b6
19d84d4c23065f5
Output = 93d48a50859a56aabe4d19f4517eff37494959536381b8c072f7d968335
dae63d62b98b448160be9559c680a4bb2aee54acd8f26fd2ce6d6e7f714124a5c56b
5
```

A.5.2.3. Test Vector 3, Batch Size 2

```
Input = 00,5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 01c6cf092d80c7cf2cb55388d899515238094c800bdd9c65f71780ba85f5
ae9b4703e17e559ca3ccd1944f9a70536c175f11a827452672b60d4e9f89eba28104
6e29,00cba1ba1a337759061965a423d9d3d6e1e1006dc8984ad28a4c93ecfc36fc2
171046b3c4284855cfa2434ed98db9e68a597db2c14728fade716a6a82d600444b26
e
BlindedElement = 020043edd4cc7f6131931455923f988620f03077d89cfc51c65
bb2f026312352dd9ccb8df17184cf8e5b3a08cfd12e12beaecbabc82b2a8c3ecf16b
e13ff4996b48e47,0300c958264a3308f411c69561ad13a61581c4d4ffecc3feff4c
0b5041b86a4a2b1b40b36a9d71ad5f0bde53514b06484fb0536dd601e42380580514
d9b7399249b406
EvaluationElement = 020011583e8c7d713ed23d2f6c2261923ef30a7534593b10
26511914b97f91a590cf50599901af6a86892abcd433b5073d869d7b330b9f4273d5
b0c614275cf0bf0c4e,02003ab9ffa4f5331b53a208999d1200b870683a53a065b3d
e57bad18a02f206968854c1d5bb94e3a7f946e3f3a6313ef2b6a9327ab160d7d2637
8dd3a880a1f002f6f
EvaluationProofC = 01dff218da0d67f76a979a4c3677a67d59a174e8e122f3d13
c0494611fe1c766d2ea08fee354ef3c7db3a6b3a3d7d75ea6a1f7e1deff35bccf27a
43ddf33ab1bf4aa
EvaluationProofS = 00434d173267092e93912d2d2714a357bb570f3105fae0191
e33e19a5eb9bb1656f361c533770fc90dc7ecaec88d0e6855c525118a1f857e7abbf
46d6b159dc4f271
Output = 64b90ccb524dc125290bda4f6102871a8580d9cb0dfe1c0e46ae87d70a1
690ddf7ed5d970ac3b51e959a8769fd8e5ee5a92ebf4e74db6104fc4334258b128d5
0,93d48a50859a56aabe4d19f4517eff37494959536381b8c072f7d968335dae63d6
2b98b448160be9559c680a4bb2aee54acd8f26fd2ce6d6e7f714124a5c56b5
```

Authors' Addresses

Alex Davidson
Brave Software

Email: alex.davidson92@gmail.com

Armando Faz-Hernandez
Cloudflare
101 Townsend St
San Francisco,
United States of America

Email: armfazh@cloudflare.com

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

Email: nick@cloudflare.com

Christopher A. Wood
Cloudflare
101 Townsend St
San Francisco,
United States of America

Email: caw@heapingbits.net