Workgroup: Network Working Group
Internet-Draft: draft-irtf-cfrg-voprf-09
Published: 8 February 2022
Intended Status: Informational
Expires: 12 August 2022
Authors: A. Davidson
Brave Software
A. Faz-Hernandez
Cloudflare, Inc.
N. Sullivan
Cloudflare, Inc.
C.A. Wood
Cloudflare, Inc.

Oblivious Pseudorandom Functions (OPRFs) using Prime-Order Groups

Abstract

An Oblivious Pseudorandom Function (OPRF) is a two-party protocol between client and server for computing the output of a Pseudorandom Function (PRF). The server provides the PRF secret key, and the client provides the PRF input. At the end of the protocol, the client learns the PRF output without learning anything about the PRF secret key, and the server learns neither the PRF input nor output. An OPRF can also satisfy a notion of 'verifiability', called a VOPRF. A VOPRF ensures clients can verify that the server used a specific private key during the execution of the protocol. A VOPRF can also be partially-oblivious, called a POPRF. A POPRF allows clients and servers to provide public input to the PRF computation. This document specifies an OPRF, VOPRF, and POPRF instantiated within standard 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

This Internet-Draft will expire on 12 August 2022.

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 also proves to the client that F(k, x) was produced by the key k corresponding to the server's public key the client knows. A Partially-Oblivious PRF (POPRF) is a variant of a VOPRF wherein client and server interact in computing F(k, x, y), for some PRF F with server-provided key k, client-provided input x, and public input y, and client receives proof that F(k, x, y) was computed using k corresponding to the public key that the client knows. A POPRF with fixed input y is functionally equivalent to a VOPRF.

OPRFs have a variety of applications, including: password-protected secret sharing schemes [JKKX16], privacy-preserving password stores [SJKS17], and password-authenticated key exchange or PAKE [I-D.irtf-cfrg-opaque]. Verifiable POPRFs are necessary in some applications such as Privacy Pass [I-D.ietf-privacypass-protocol]. Verifiable POPRFs have also been used for password-protected secret sharing schemes such as that of [JKK14].

This document specifies OPRF, VOPRF, and POPRF protocols built upon prime-order groups. The document describes each protocol variant, along with application considerations, and their security properties.

1.1. Change log

draft-09:

- *Split syntax for OPRF, VOPRF, and POPRF functionalities.
- *Make Blind function fallible for invalid private and public inputs.
- *Specify key generation.
- *Remove serialization steps from core protocol functions.
- *Refactor protocol presentation for clarity.
- *Simplify security considerations.
- *Update application interface considerations.
- *Update test vectors.

draft-08:

- *Adopt partially-oblivious PRF construction from [TCRSTW21].
- *Update P-384 suite to use SHA-384 instead of SHA-512.

- *Update test vectors.

- *Apply various editorial changes.

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.

1.3. Notation and Terminology

The following functions and notation 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.

All algorithms and procedures described in this document are laid out in a Python-like pseudocode. The data types PrivateInput and PublicInput are opaque byte strings of arbitrary length no larger than 2^13 octets.

String values such as "Finalize" are ASCII string literals.

The following terms are used throughout this document.

* PRF: Pseudorandom Function.

* OPRF: Oblivious Pseudorandom Function.

* VOPRF: Verifiable Oblivious Pseudorandom Function.

* POPRF: Partially Oblivious Pseudorandom Function.

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

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

2. Preliminaries

The protocols in this document have two primary dependencies:

*Group: 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 whose output length is Nh bytes long.

Section 4 specifies ciphersuites as combinations of Group and Hash.

2.1. Prime-Order Group

In this document, we assume the construction of an additive, prime-order group Group for performing all mathematical operations. Such groups are uniquely determined by the choice of the prime p that defines the order of the group. (There may, however, exist different representations of the group for a single p. Section 4 lists specific groups which indicate both order and representation.) 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, A + B = B + A is also a member of the group. Also, for any A in the group, 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. Scalar base multiplication is equivalent to the repeated application of the group operation on the fixed group generator with itself r-1 times, and 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 and its set of scalars, respectively.

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

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

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

*HashToGroup(x): A member function of Group that deterministically maps an array of bytes x to an element of Group. 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 Group 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 Group that chooses at random a non-zero element in GF(p).

- ScalarInverse(s): Compute the multiplicative inverse of input Scalar s modulo the prime order of the group p.

- SerializeElement(A): A member function of Group 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 Group 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 4.2.

- SerializeScalar(s): A member function of Group 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 Group 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 4.2.

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. Discrete Log Equivalence Proofs

Another important piece of the OPRF protocols in this document is proving that the discrete log of two values is identical in zero knowledge, i.e., without revealing the discrete logarithm. This is referred to as a discrete log equivalence (DLEQ) proof. This section describes functions for non-interactively proving and verifying this type of statement, built on a Chaum-Pedersen [ChaumPedersen] proof. It is split into two sub-sections: one for generating the proof, which is done by servers in the verifiable protocols, and another for verifying the proof, which is done by clients in the protocol.

2.2.1. Proof Generation

Generating a proof is done with the GenerateProof function, defined below. This function takes four Elements, A, B, C, and D, and a single group Scalar k, and produces a proof that k*A == B and k*C == D. The output is a value of type Proof, which is a tuple of two Scalar values.

```
Input:

  Scalar k
  Element A
  Element B
  Element C
  Element D

Output:

  Proof proof

Parameters:

  Group G

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

  r = G.RandomScalar()
  t2 = r * A
  t3 = r * M

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

  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 ||
            "Challenge"

  c = G.HashToScalar(h2Input)
  s = (r - c * k) mod G.Order()

  return [c, s]
```

The helper function ComputeCompositesFast is as defined below.

```
Input:

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

Output:

  Element M
  Element Z

Parameters:

  Group G
  PublicInput contextString

def ComputeCompositesFast(k, B, Cs, Ds):
  Bm = G.SerializeElement(B)
  seedDST = "Seed-" || contextString
  h1Input = I2OSP(len(Bm), 2) || Bm ||
            I2OSP(len(seedDST), 2) || seedDST
  seed = Hash(h1Input)

  M = G.Identity()
  for i = 0 to range(m):
    Ci = G.SerializeElement(Cs[i])
    Di = G.SerializeElement(Ds[i])
    h2Input = I2OSP(len(seed), 2) || seed || I2OSP(i, 2) ||
              I2OSP(len(Ci), 2) || Ci ||
              I2OSP(len(Di), 2) || Di ||
              "Composite"

    di = G.HashToScalar(h2Input)
    M = di * Cs[i] + M

  Z = k * M

 return (M, Z)
```

When used in the protocol described in Section 3, the parameter contextString is as defined in Section 3.1.

ComputeCompositesFast 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 particular, servers can produce a single, constant-sized proof for N DLEQ inputs, rather than one proof per DLEQ input. This optimization benefits clients and servers since it

amortizes the cost of proof generation and bandwidth across multiple requests.

2.2.2. Proof Verification

Verifying a proof is done with the VerifyProof function, defined below. This function takes four Elements, A, B, C, and D, along with a Proof value output from GenerateProof. It outputs a single boolean value indicating whether or not the proof is valid for the given DLEQ inputs.

```
Input:

  Element A
  Element B
  Element C
  Element D
  Proof proof

Output:

  boolean verified

Parameters:

  Group G

Errors: DeserializeError

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

  (M, Z) = ComputeComposites(B, Cs, Ds)
  c = proof[0]
  s = proof[1]

  t2 = ((s * A) + (c * B))
  t3 = ((s * M) + (c * Z))

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

  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 ||
            "Challenge"

  expectedC = G.HashToScalar(h2Input)

  return expectedC == c
```

The definition of ComputeComposites is given below.

```
Input:

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

Output:

  Element M
  Element Z

Parameters:

  Group G
  PublicInput contextString

def ComputeComposites(B, Cs, Ds):
  Bm = G.SerializeElement(B)
  seedDST = "Seed-" || contextString
  h1Input = I2OSP(len(Bm), 2) || Bm ||
            I2OSP(len(seedDST), 2) || seedDST
  seed = Hash(h1Input)

  M = G.Identity()
  Z = G.Identity()
  for i = 0 to m-1:
    Ci = G.SerializeElement(Cs[i])
    Di = G.SerializeElement(Ds[i])
    h2Input = I2OSP(len(seed), 2) || seed || I2OSP(i, 2) ||
              I2OSP(len(Ci), 2) || Ci ||
              I2OSP(len(Di), 2) || Di ||
              "Composite"

    di = G.HashToScalar(h2Input)
    M = di * Cs[i] + M
    Z = di * Ds[i] + Z

 return (M, Z)
```

When used in the protocol described in Section 3, the parameter contextString is as defined in Section 3.1.

As with the proof generation case, proof verification can be batched. ComputeComposites is defined in terms of a batch of inputs. Implementations can take advantage of this behavior by also batching inputs to VerifyProof, respectively.

3. Protocol

In this section, we define three OPRF protocol variants -- a base mode, verifiable mode, and partially-oblivious mode -- with the following properties.

In the base mode, a client and server interact to compute output = F(skS, input), where input is the client's private input, skS is the server's private key, and output is the OPRF output. The client learns output and the server learns nothing. This interaction is shown below.

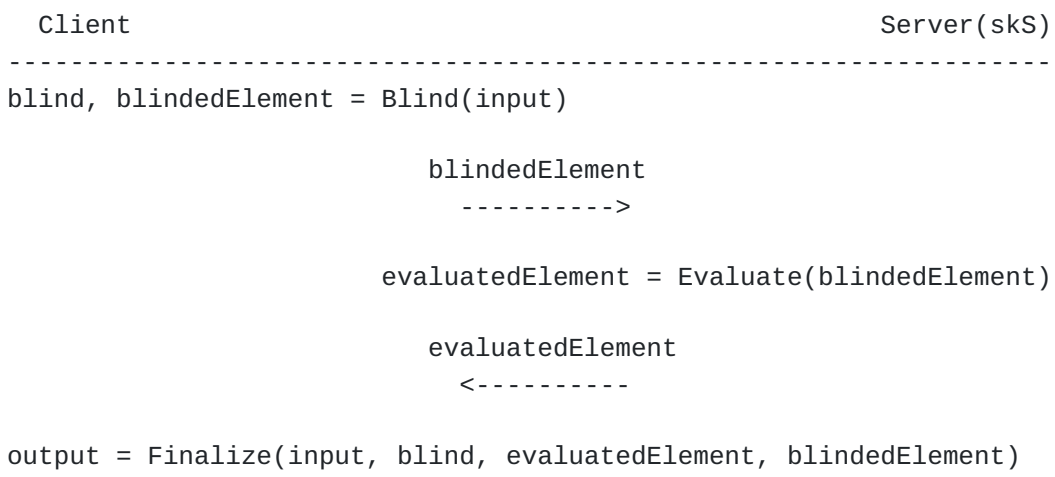

Figure 1: OPRF protocol overview

In the verifiable mode, the client additionally receives proof that the server used skS in computing the function. To achieve verifiability, as in the original work of [JKK14], the server provides 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 the server's public key, pkS, which the client receives as input to the protocol. This proof does not reveal the server's private key to the client. This interaction is shown below.

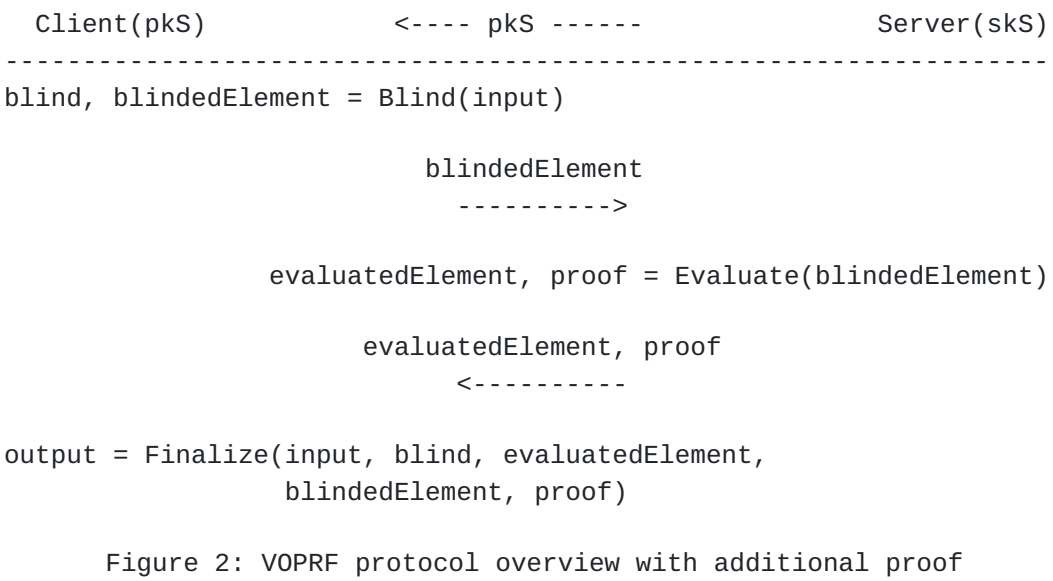

Figure 2: VOPRF protocol overview with additional proof

The partially-oblivious mode extends the VOPRF mode such that the client and server can additionally provide a public input info that is used in computing the pseudorandom function. That is, the client and server interact to compute output = F(skS, input, info). To support additional public input, the client and server augment the pkS and skS, respectively, using the info value, as in [TCRSTW21].

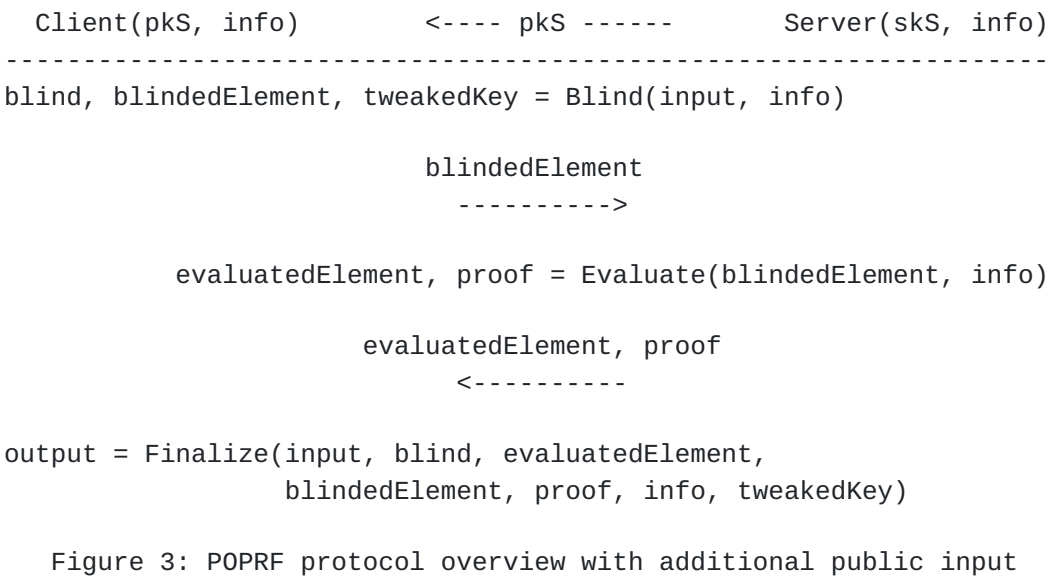

Figure 3: POPRF protocol overview with additional public input

Each protocol consists of an offline setup phase and an online phase, described in Section 3.2 and Section 3.3, respectively. Configuration details for the offline phase are described in Section 3.1.

3.1. Configuration

Each of the three protocol variants are identified with a one-byte value:

Mode	Value
modeOPRF	0x00
modeVOPRF	0x01
modePOPRF	0x02

Table 1

Additionally, each protocol variant is instantiated with a ciphersuite, or suite. Each ciphersuite is identified with a two-byte value, referred to as suiteID; see Section 4 for the registry of initial values.

The mode and ciphersuite ID values are combined to create a "context string" used throughout the protocol with the following function:

```
def CreateContextString(mode, suiteID):
  return "VOPRF09-" || I2OSP(mode, 1) || I2OSP(suiteID, 2)
```

[[RFC editor: please change "VOPRF09" to "RFCXXXX", where XXXX is the final number, here and elsewhere before publication.]]

3.2. Key Generation and Context Setup

In the offline setup phase, both the client and server create a context used for executing the online phase of the protocol after agreeing on a mode and ciphersuite value suiteID. The server key pair (skS, pkS) is generated using the following function, which accepts a randomly generated seed of length Ns and optional public info string. The constant Ns corresponds to the size of a serialized Scalar and is defined in Section 2.1.

```
Input:

  opaque seed[Ns]
  PublicInput info

Output:

  Scalar skS
  Element pkS

Parameters:

  Group G
  PublicInput contextString

Errors: DeriveKeyPairError

def DeriveKeyPair(seed, info):
  contextString = CreateContextString(mode, suiteID)
  deriveInput = seed || I2OSP(len(info), 2) || info
  counter = 0
  skS = 0
  while skS == 0:
    if counter > 255:
      raise DeriveKeyPairError
    skS = G.HashToScalar(deriveInput || I2OSP(counter, 1),
                         DST = "DeriveKeyPair" || contextString)
    counter = counter + 1
  pkS = G.ScalarBaseMult(skS)
  return skS, pkS
```

The OPRF variant server and client contexts are created as follows:

```
def SetupOPRFServer(suiteID, skS):
  contextString = CreateContextString(modeOPRF, suiteID)
  return OPRFServerContext(contextString, skS)

def SetupOPRFClient(suiteID):
  contextString = CreateContextString(modeOPRF, suiteID)
  return OPRFClientContext(contextString)
```

The VOPRF variant server and client contexts are created as follows:

```
def SetupVOPRFServer(suiteID, skS, pkS):
  contextString = CreateContextString(modeVOPRF, suiteID)
  return VOPRFServerContext(contextString, skS)

def SetupVOPRFClient(suiteID, pkS):
  contextString = CreateContextString(modeVOPRF, suiteID)
  return VOPRFClientContext(contextString, pkS)
```

The POPRF variant server and client contexts are created as follows:

```
def SetupPOPRFServer(suiteID, skS, pkS):
  contextString = CreateContextString(modePOPRF, suiteID)
  return POPRFServerContext(contextString, skS)

def SetupPOPRFClient(suiteID, pkS):
  contextString = CreateContextString(modePOPRF, suiteID)
  return POPRFClientContext(contextString, pkS)
```

3.3. Online Protocol

In the online phase, the client and server engage in a two message protocol to compute the protocol output. This section describes the protocol details for each protocol variant. Throughout each description the following parameters are assumed to exist:

- G, a prime-order Group implementing the API described in Section 2.1.
- contextString, a PublicInput domain separation tag constructed during context setup as created in Section 3.1.
- skS and pkS, a Scalar and Element representing the private and public keys configured for client and server in Section 3.2.

Applications serialize protocol messages between client and server for transmission. Specifically, values of type Element are serialized to SerializedElement values, and values of type Proof are serialized as the concatenation of two SerializedScalar values. Deserializing these values can fail, in which case the application MUST abort the protocol with a DeserializeError failure.

Applications MUST check that input Element values received over the wire are not the group identity element. This check is handled when deserializing Element values using DeserializeElement; see Section 4.2 for more information on input validation.

3.3.1. OPRF Protocol

The OPRF protocol begins with the client blinding its input, as described by the Blind function below. Note that this function can fail with an InvalidInputError error for certain inputs that map to the group identity element. Dealing with this failure is an application-specific decision; see Section 5.3.

```
Input:

  PrivateInput input

Output:

  Scalar blind
  Element blindedElement

Parameters:

  Group G

Errors: InvalidInputError

def Blind(input):
  blind = G.RandomScalar()
  P = G.HashToGroup(input)
  if P == G.Identity():
    raise InvalidInputError
  blindedElement = blind * P

  return blind, blindedElement
```

Clients store blind locally, and send blindedElement to the server for evaluation. Upon receipt, servers process blindedElement using the Evaluate function described below.

```
Input:

  Element blindedElement

Output:

  Element evaluatedElement

Parameters:

  Scalar skS

def Evaluate(blindedElement):
  evaluatedElement = skS * blindedElement
  return evaluatedElement
```

Servers send the output evaluatedElement to clients for processing. Recall that servers may batch multiple client inputs to Evaluate.

Upon receipt of evaluatedElement, clients process it to complete the OPRF evaluation with the Finalize function described below.

```
Input:

  PrivateInput input
  Scalar blind
  Element evaluatedElement

Output:

  opaque output[Nh]

Parameters:

  Group G

def Finalize(input, blind, evaluatedElement):
  N = G.ScalarInverse(blind) * evaluatedElement
  unblindedElement = G.SerializeElement(N)

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

3.3.2. VOPRF Protocol

The VOPRF protocol begins with the client blinding its input, using the same Blind function as in Section 3.3.1. Clients store the output blind locally and send blindedElement to the server for evaluation. Upon receipt, servers process blindedElement to compute an evaluated element and DLEQ proof using the following Evaluate function.

```
Input:

  Element blindedElement

Output:

  Element evaluatedElement
  Proof proof

Parameters:

  Group G
  Scalar skS
  Element pkS

def Evaluate(blindedElement):
  evaluatedElement = skS * blindedElement
  proof = GenerateProof(skS, G.Generator(), pkS,
                        blindedElement, evaluatedElement)
  return evaluatedElement, proof
```

The server sends both evaluatedElement and proof back to the client. Upon receipt, the client processes both values to complete the VOPRF computation using the Finalize function below.

```
Input:

  PrivateInput input
  Scalar blind
  Element evaluatedElement
  Element blindedElement
  Proof proof

Output:

  opaque output[Nh]

Parameters:

  Group G
  Element pkS

Errors: VerifyError

def Finalize(input, blind, evaluatedElement, blindedElement, proof):
  if VerifyProof(G.Generator(), pkS, blindedElement,
                 evaluatedElement, proof) == false:
    raise VerifyError

  N = G.ScalarInverse(blind) * evaluatedElement
  unblindedElement = G.SerializeElement(N)

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

3.3.3. POPRF Protocol

The POPRF protocol begins with the client blinding its input, using the following modified Blind function. Note that this function can fail with an InvalidInputError error for certain private inputs that map to the group identity element, as well as certain public inputs that map to invalid public keys for server evaluation. Dealing with either failure is an application-specific decision; see Section 5.3.

```
Input:

  PrivateInput input
  PublicInput info

Output:

  Scalar blind
  Element blindedElement
  Element tweakedKey

Parameters:

  Group G
  Element pkS

Errors: InvalidInputError

def Blind(input, info):
  framedInfo = "Info" || I2OSP(len(info), 2) || info
  m = G.HashToScalar(framedInfo)
  T = G.ScalarBaseMult(m)
  tweakedKey = T + pkS
  if tweakedKey == G.Identity():
    raise InvalidInputError

  blind = G.RandomScalar()
  P = G.HashToGroup(input)
  if P == G.Identity():
    raise InvalidInputError

  blindedElement = blind * P

  return blind, blindedElement, tweakedKey
```

Clients store the outputs blind and tweakedKey locally and send blindedElement to the server for evaluation. Upon receipt, servers process blindedElement to compute an evaluated element and DLEQ proof using the following Evaluate function.

```
Input:

  Element blindedElement
  PublicInput info

Output:

  Element evaluatedElement
  Proof proof

Parameters:

  Group G
  Scalar skS
  Element pkS

Errors: InverseError

def Evaluate(blindedElement, info):
  framedInfo = "Info" || I2OSP(len(info), 2) || info
  m = G.HashToScalar(framedInfo)
  t = skS + m
  if t == 0:
    raise InverseError

  evaluatedElement = G.ScalarInverse(t) * blindedElement

  tweakedKey = G.ScalarBaseMult(t)
  proof = GenerateProof(t, G.Generator(), tweakedKey,
                        evaluatedElement, blindedElement)

  return evaluatedElement, proof
```

The server sends both evaluatedElement and proof back to the client. Upon receipt, the client processes both values to complete the VOPRF computation using the Finalize function below.

```
Input:

  PrivateInput input
  Scalar blind
  Element evaluatedElement
  Element blindedElement
  Proof proof
  PublicInput info
  Element tweakedKey

Output:

  opaque output[Nh]

Parameters:

  Group G
  Element pkS

Errors: VerifyError

def Finalize(input, blind, evaluatedElement, blindedElement,
             proof, info, tweakedKey):
  if VerifyProof(G.Generator(), tweakedKey, evaluatedElement,
                 blindedElement, proof) == false:
    raise VerifyError

  N = G.ScalarInverse(blind) * evaluatedElement
  unblindedElement = G.SerializeElement(N)

  hashInput = I2OSP(len(input), 2) || input ||
              I2OSP(len(info), 2) || info ||
              I2OSP(len(unblindedElement), 2) || unblindedElement ||
              "Finalize"
  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. The 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:

- Group: 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 whose output length is Nh bytes long.

This section specifies an initial registry of ciphersuites with supported groups and hash functions. It also includes implementation details for each ciphersuite, focusing on input validation, as well as requirements for future ciphersuites.

4.1. Ciphersuite Registry

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.

4.1.1. 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

4.1.2. 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

4.1.3. 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

4.1.4. OPRF(P-384, SHA-384)

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

-HashToGroup(): Use hash_to_curve with suite P384_XMD:SHA-384_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-384, 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-384, and Nh = 48.

*ID: 0x0004

4.1.5. 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

4.2. Input Validation

The DeserializeElement and DeserializeScalar functions instantiated for a particular prime-order group corresponding to a ciphersuite MUST adhere to the description in Section 2.1. This section describes how both DeserializeElement and DeserializeScalar are

implemented for all prime-order groups included in the above ciphersuite list.

4.2.1. DeserializeElement Validation

The DeserializeElement function attempts to recover 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 input is invalid. If this function returns false, deserialization returns an error.

4.2.2. DeserializeScalar Validation

The DeserializeScalar function attempts to recover 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() - 1. For ristretto255 and decaf448, this function ensures that the input, when treated as a little-endian integer, is a value between 0 and Order() - 1.

4.3. Future Ciphersuites

A critical requirement of implementing the prime-order group using elliptic curves is a method to instantiate the function 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 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.

Additionally, future ciphersuites must take care when choosing the security level of the group. See Section 6.2.3 for additional details.

5. Application Considerations

This section describes considerations for applications, including external interface recommendations, explicit error treatment, and public input representation for the POPRF protocol variant.

5.1. Input Limits

Application inputs, expressed as PrivateInput or PublicInput values, MUST be smaller than 2^13 bytes in length. Applications that require longer inputs can use a cryptographic hash function to map these longer inputs to a fixed-length input that fits within the PublicInput or PrivateInput length bounds. Note that some cryptographic hash functions have input length restrictions themselves, but these limits are often large enough to not be a concern in practice. For example, SHA-256 has an input limit of 2^61 bytes.

5.2. External Interface Recommendations

The protocol functions in Section 3.3 are specified in terms of prime-order group Elements and Scalars. However, applications can treat these as internal functions, and instead expose interfaces that operate in terms of wire format messages.

5.3. Error Considerations

Some OPRF variants specified in this document have fallible operations. 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: Verifiable OPRF proof verification failed; Section 3.3.2 and Section 3.3.3.

- DeserializeError: Group Element or Scalar deserialization failure; Section 2.1 and Section 3.3.

There are other explicit errors generated in this specification, however they occur with negligible probability in practice. We note them here for completeness.

- *InvalidInputError: OPRF Blind input produces an invalid output element; Section 3.3.1 and Section 3.3.3.

- *InverseError: A tweaked private key is invalid (has no multiplicative inverse); Section 2.1 and Section 3.3.

In general, 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.

5.4. POPRF Public Input

Functionally, the VOPRF and POPRF variants differ in that the POPRF variant admits public input, whereas the VOPRF variant does not. Public input allows clients and servers to cryptographically bind additional data to the POPRF output. A POPRF with fixed public input is functionally equivalent to a VOPRF. However, there are differences in the underlying security assumptions made about each variant; see Section 6.2 for more details.

This public input is known to both parties at the start of the protocol. It is RECOMMENDED that this public input be constructed with some type of higher-level domain separation to avoid cross protocol attacks or related issues. For example, protocols using this construction might ensure that the public input uses a unique, prefix-free encoding. See [I-D.irtf-cfrg-hash-to-curve], Section 10.4 for further discussion on constructing domain separation values.

Implementations of the POPRF may choose to not let applications control info in cases where this value is fixed or otherwise not useful to the application. In this case, the resulting protocol is functionally equivalent to the VOPRF, which does not admit public input.

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 the OPRF variants in this document. Note that the syntax of the POPRF variant is different from that of the OPRF and POPRF variants since it admits an additional public input, but the same security considerations apply.

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.

- Unconditional input secrecy: The server does not learn anything about the client input x, even with unbounded computation.

In other words, an attacker with infinite compute cannot recover any information about the client's private input x from an invocation of the protocol.

Additionally, for the VOPRF and POPRF protocol variants, there is an additional security property:

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

Any VOPRF or POPRF that satisfies the 'verifiable' security property is known as 'verifiable'. 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.

Finally, the POPRF variant also has the following security property:

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

private key. Both client and server learn the public input (info).

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

6.2. Security Assumptions

Below, we discuss the cryptographic security of each protocol variant from Section 3, relative to the necessary cryptographic assumptions that need to be made.

6.2.1. OPRF and VOPRF Assumptions

The OPRF and VOPRF protocol variants in this document are based on [JKK14]. In fact, the VOPRF construction is identical to the [JKK14] construction, except that this document supports batching so that multiple evaluations can happen at once whilst only constructing one proof object. This is enabled using an established batching technique.

The pseudorandomness and input secrecy (and verifiability) of the OPRF (and VOPRF) variants is based on the assumption that the One-More Gap Computational Diffie Hellman (CDH) is computationally difficult to solve in the corresponding prime-order group. The original paper [JKK14] gives a security proof that the construction satisfies the security guarantees of a VOPRF protocol Section 6.1 under the One-More Gap CDH assumption in the universal composability (UC) security framework.

6.2.2. POPRF Assumptions

The POPRF construction in this document is based on the construction known as 3HashSDHI given by [TCRSTW21]. The construction is identical to 3HashSDHI, except that this design can optionally perform multiple POPRF evaluations in one go, whilst only constructing one NIZK proof object. This is enabled using an established batching technique.

Pseudorandomness, input secrecy, verifiability, and partial obliviousness of the POPRF variant is based on the assumption that the One-More Gap Strong Diffie-Hellman Inversion (SDHI) assumption from [TCRSTW21] is computationally difficult to solve in the corresponding prime-order group. [TCRSTW21] show that both the One-More Gap CDH assumption and the One-More Gap SDHI assumption reduce to the q-DL (Discrete Log) assumption in the algebraic group model, for some q number of Evaluate queries. (The One-More Gap CDH

assumption was the hardness assumption used to evaluate the OPRF and VOPRF designs based on [JKK14], which is a predecessor to the POPRF variant in Section 3.3.3.)

6.2.3. Static Diffie Hellman Attack and Security Limits

A side-effect of the OPRF protocol variants in this document is that they allow instantiation of an oracle for constructing static DH samples; see [BG04] and [Cheon06]. These attacks are meant to recover (bits of) the server private key. Best-known attacks reduce the security of the prime-order group instantiation by log_2(Q)/2 bits, where Q is the number of Evalute() calls made by the attacker.

As a result of this class of attack, choosing prime-order groups with a 128-bit security level instantiates an OPRF with a reduced security level of 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. Applications can mitigate against this problem in a variety of ways, e.g., by rate-limiting client queries to Evaluate() or by rotating private keys. 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. Applications that admit the aforementioned oracle functionality, and that cannot tolerate discrete logarithm security of lower than 128 bits, are RECOMMENDED to choose groups that target a higher security level, such as decaf448 (used by ciphersuite 0x0002), P-384 (used by 0x0004), or P-521 (used by 0x0005).

6.3. Domain Separation

Applications SHOULD construct input to the protocol to provide domain separation. Any system which has multiple 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. Timing Leaks

To ensure no information is leaked during protocol execution, all operations that use secret data MUST run in constant time. This includes all prime-order group operations and proof-specific operations that operate on secret data, including GenerateProof() and Evaluate().

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, Kevin Lewi, Christopher Patton, and Bas Westerbaan also provided helpful input and contributions to the document.

8. References

8.1. Normative References

[I-D.ietf-privacypass-protocol] Celi, S., Davidson, A., Faz-Hernandez, A., Valdez, S., and C. A. Wood, "Privacy Pass Issuance Protocol", Work in Progress, Internet-Draft, draft-ietf-privacypass-protocol-02, 31 January 2022, <https://www.ietf.org/archive/id/draft-ietf-privacypass-protocol-02.txt>.

[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-13, 10 November 2021, <https://www.ietf.org/archive/id/draft-irtf-cfrg-hash-to-curve-13.txt>.

[I-D.irtf-cfrg-opaque] Bourdrez, D., Krawczyk, H., Lewi, K., and C. A. Wood, "The OPAQUE Asymmetric PAKE Protocol", Work in Progress, Internet-Draft, draft-irtf-cfrg-opaque-07, 25 October 2021, <https://www.ietf.org/archive/id/draft-irtf-cfrg-opaque-07.txt>.

[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>.

[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/info/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/info/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-decaf448-01, 4 August 2021, <https://www.ietf.org/archive/id/draft-irtf-cfrg-ristretto255-decaf448-01.txt>.

8.2. Informative References

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

[ChaumPedersen] "Wallet Databases with Observers", n.d., <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>.

[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>.

[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>.

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

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

[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>.

[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>.

[TCRSTW21] "A Fast and Simple Partially Oblivious PRF, with Applications", <https://eprint.iacr.org/2021/864>.

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

Appendix A. Test Vectors

This section includes test vectors for the protocol variants specified in this document. For each ciphersuite specified in Section 4, there is a set of test vectors for the protocol when run the OPRF, VOPRF, and POPRF modes. 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 private client input, an opaque byte string.

* "Info": The public info, an opaque byte string. Only present for POPRF vectors.

* "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.

* "Proof": The serialized Proof output from GenerateProof() (only listed for verifiable mode test vectors), composed of two serialized Scalar values each of Ns bytes long. Only present for VOPRF and POPRF vectors.

* "ProofRandomScalar": The random scalar r computed in GenerateProof() (only listed for verifiable mode test vectors), a serialized Scalar of Ns bytes long. Only present for VOPRF and POPRF vectors.

* "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.

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 Seed and info string KeyInfo, which are listed as well, using the DeriveKeyPair function from Section 3.2.

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

A.1.1. OPRF Mode

```
Seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3
3a3
KeyInfo = 74657374206b6579
skSm = 8ce0798c296bdeb665d52312d81a596dbb4ef0d25adb10c7f2b58c72dd2e5
40a
```

A.1.1.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = c604c785ada70d77a5256ae21767de8c3304115237d262134f5e46e512cf
8e03
BlindedElement = 8453ce4f98478a73faf24dd0c2e81d9a5e399171d2687cc258b
9e593623bde4d
EvaluationElement = 22bcfc0930ecddf4ada3f0cb421c8d6669576fc4fbbe24e1
8c94d0f36e767466
Output = 2765a7f9fa7e9d5440bbf1262dc1041277bed5f27fd27ee89662192a408
508bb8711559d5a5390560065b83b946ed7b433d0c1df09bd23871804ae78e4a4d21
5
```

A.1.1.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 5ed895206bfc53316d307b23e46ecc6623afb3086da74189a416012be037
e50b
BlindedElement = 86ef8baa01dd6cc34a067d2fc56cde51498a54cb0c30f63f083
53d912164d711
EvaluationElement = a27d5e498927ca96e493373a04e263115c31b918411df0ce
d382db4e66388766
Output = 3d6c9ec7dd6f51b987b46b79128d98323accd7c1561faa50d287c5285ec
da1e660f3ee2929aebd431a7a7d511767cbd1054735a6e19aee1b9423a1e6f479535
e
```

A.1.2. VOPRF Mode

```
Seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3
3a3
KeyInfo = 74657374206b6579
skSm = 24d1aca670a768b99e888c19f9ee3252d17e32689507b2b9b803ae23b1997
f07
pkSm = 46919d396c12fbb7a02f4ce8f51a9941ddc1c4335682e1b03b0ca5b3524c6
619
```

A.1.2.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = ed8366feb6b1d05d1f46acb727061e43aadfafe9c10e5a64e7518d63e326
3503
BlindedElement = 444550ea064013c569fe63567eb93e7a9496902a573ea1e6654
76fd39d5edc40
EvaluationElement = 7af7a45e4f1e0c6d410d41704e16d980ebff051fd0975fce
cd17f79a6b57a473
Proof = 26982a26b2aa20f1e449be5a858c59d7992f7f4a13b007e3980f5c36e8ae
a7014268883db3094e08e3f493b3d23bae87ac098a33e775172c1027f1b5d025ca08
ProofRandomScalar = 019cbd1d7420292528f8cdd62f339fdabb602f04a95dac9d
bcec831b8c681a09
Output = 453a358544b4e92bbc4625d08ffdde64c0dbc4f9b1501d548e3a6d8094b
a70a993c13a6e65a46880bbd65272ba54cf199577760815098e5e10cb951b1fc5b02
7
```

A.1.2.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = e6d0f1d89ad552e383d6c6f4e8598cc3037d6e274d22da3089e7afbd4171
ea02
BlindedElement = 82d9fc20daf67106ae2d2c584c615d103dafda653ac5b2b2c6f
aafc06e3f1c0a
EvaluationElement = 60c468920f4f949be9aaaf9b4fb27dc7bc89daca4a3aaa31
e96efae56c02ac75
Proof = 4a7490fd0a9e13cc66bcdeded002899a3e206364d9bdbaf9998a73dd728c
8602a6967f81a4948e6de797d638ee02ca44d933d05f2715fa1618b6a3324f3b2608
ProofRandomScalar = 74ae06fd50d5f26c2519bd7b184f45dd3ef2cb50197d42df
9d013f7d6c312a0b
Output = a2bacfc82a4cac041edab1e1c0d0dc63f46631fb4886f8c395f0b184a9b
7cbbef2eee05bbd3f085552d8c80e77711b2ad9ba2b7574e2531591380e717d29c6f
5
```

A.1.2.3. Test Vector 3, Batch Size 2

```
Input = 00,5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 80513e77795feeec6d2c450589b0e1b178febd5c193a9fcba0d27f0a06e0
d50f,533c2e6d91c934f919ac218973be55ba0d7b234160a0d4cf3bddafbda99e2e0
c
BlindedElement = 70a6ac589da4cfff4a1135c21e438a50935317ad6900810a59e
76c2c28d8e562,5ed4710468c94e6c0181aef8276204ec6aef509f5cf1d7d6184693
1481d23d76
EvaluationElement = 5ef7bc4c54aa5fccb4328fd725d3c20130ebe3ced54f28b6
e6c4591815158059,0ce1a236be8dba445cf57ddddec8f1c2d9be2c164add431fc18
e3279be968c2d
Proof = 53afba40c6c27636a0694def258728f192d25ec5f97ee1e87a408fd20615
6107d3b82b618242f10ff459d7d30d0a68d9e381254d2e5f6bc82671f093f47c0e01
ProofRandomScalar = 3af5aec325791592eee4a8860522f8444c8e71ac33af5186
a9706137886dce08
Output = 453a358544b4e92bbc4625d08ffdde64c0dbc4f9b1501d548e3a6d8094b
a70a993c13a6e65a46880bbd65272ba54cf199577760815098e5e10cb951b1fc5b02
7,a2bacfc82a4cac041edab1e1c0d0dc63f46631fb4886f8c395f0b184a9b7cbbef2
eee05bbd3f085552d8c80e77711b2ad9ba2b7574e2531591380e717d29c6f5
```

A.1.3. POPRF Mode

```
Seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3
KeyInfo = 74657374206b6579
skSm = 02c1b949c3d65f83b18890aa8d099f6063fa6a72add8d776bc15a291fd08f
f04
pkSm = 1a2fac6b919790c613e0fbed070471778fbb1c6950d40a4e059acb652dc57
161
```

A.1.3.1. Test Vector 1, Batch Size 1

```
Input = 00
Info = 7465737420696e666f
Blind = 7e5bcbf82a46109ee0d24e9bcab41fc830a6ce8b82fc1e9213a043b743b9
5800
BlindedElement = da01485047605a666542d0599ef2fbeed0c2e45a97c6e3d420f
832918e09f535
EvaluationElement = 3015fc16fe179bdb9054da5297c77d1f249dabf32e4fdcc4
937d6ba5e99d7b53
Proof = f10470180fc884a2f51472eddde9ad9a4080b00e13f63c130cece83b93ca
500f956b08e35ed2670ca504c704e0b74687451f5985627c93e2290a5da0dffc1d0b
ProofRandomScalar = 080d0a4d352de92672ab709b1ae1888cb48dfabc2d6ca5b9
14b335512fe70508
Output = 4d04eccb77a29bd8a00fb1e3f391e0601340c3dc874fc7bb16cfd92d961
532d18b4edfffaec94457cb19111bca1ecd19e46124c6a5d29703d09df5e5ab521b2
8
```

A.1.3.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Info = 7465737420696e666f
Blind = de2e98f422bf7b99be19f7da7cac62f1599d35a225ec6340149a0aaff310
2003
BlindedElement = 909f8d2d517fa2235f8b35f91220636732541d9f3e309c6988d
6d8c987e5a357
EvaluationElement = 241786b8f9da3e8c28d75dc23b5f8b251ec150ccb453efa7
12f6e9b72e763a0a
Proof = a3748b980aec81add561bcd7ac4fe2b09a93bd8a127991788fd618bf7fb7
93034a6f7f59cdcab538ed3e50d74b31f82dff14e3c8d3a081f744a6bdf93526ed0e
ProofRandomScalar = c4d002aa4cfcf281657cf36fe562bc60d9133e0e72a74432
f685b2b6a4b42a0c
Output = a88ab2bceba2c9c5a0ee0ee45636e65042b5f274af864f8c1560d32ecee
4373c31907f237609d3f164beec32e3270588961c1d19cee467d2a3b0445ebdea215
9
```

A.1.3.3. Test Vector 3, Batch Size 2

```
Input = 00,5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Info = 7465737420696e666f
Blind = e79a642b20f4c9118febffaf6b6a31471fe7794aa77ced123f07e56cb8cf
7c01,0bb106c0e1aac79e92dd2d051e90efe4e2e093bc1e82b80e8cce6afa4f51980
2
BlindedElement = 3206271954cce85425971fddfebe14acad819b9753ffc171815
7e54a5e56542f,847b21e32855892256a3eee10ea5c512d362b34de1ab278573cf91
edfcb14a03
EvaluationElement = 76d3282ac9aabc9b0133df89e680ab0d43f2946c224db25e
798abdf0ed1d255a,e8483fbacb3e62787a803dd6d688e4db26be5392f529f1dd6a7
f06e2b28dc52c
Proof = f5b8d39897f12dd1f8fc927e2f7f563629b7b45f1e6b5eeb469c043d2143
7907a0e9236beec240a04e0fb906a7d126a8cb40e22730106446c1fa3a40a5283406
ProofRandomScalar = 668b3aab5207735beb86c5379228da260159dc24f7c5c248
3a81aff8fbffcc0d
Output = 4d04eccb77a29bd8a00fb1e3f391e0601340c3dc874fc7bb16cfd92d961
532d18b4edfffaec94457cb19111bca1ecd19e46124c6a5d29703d09df5e5ab521b2
8,a88ab2bceba2c9c5a0ee0ee45636e65042b5f274af864f8c1560d32ecee4373c31
907f237609d3f164beec32e3270588961c1d19cee467d2a3b0445ebdea2159
```

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

A.2.1. OPRF Mode

```
Seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3
KeyInfo = 74657374206b6579
skSm = 416f3b0f4d13ac19e6aacade4ecf8b7e9c55d808311be2bea0dae4f4c56d0
73e7229b8b72a8c7eb68bd2e98336baaec1ac47c82cf2c5e33b
```

A.2.1.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = d62851d4bc07947c858dc735e9e22aaf0576f161ab555182908dbd7947b1
c988956fa73b17b373b72fd4e3c0264a26aa4cab20fd6193b933
BlindedElement = d078a185d2d8a54b68d6df4e83640192d3659e18fec68d43e48
02998d3c9fd819b32070caa78083c909d68daeb7fd420a73f931452a2b70d
EvaluationElement = 3452e46b6277b032627a7e5d22aa1b25459f8de90dda3137
9ed490bb0078eeec05fc4265fafbb5252d4228f9f1f5453bbd391d6b8589f232
Output = b93d3ed18489c1236cc965d202254de35767ea673560d6c225cec0b30fe
3adc88fee63f8a78d127cd64c7077e1d3ac4a7cc761335c0bcdc12d6981ad8730285
8
```

A.2.1.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = ac345e8d755997956ddd1f267a2d86175aeae5e1168932285a6f602b4b20
a570a697452b3ddbb7d0e29363adebbcb5673294396b82931f37
BlindedElement = 283f0fab2be6ac3a3c8eacfd504f3ef63f518892f7b000f1dcc
1ca2e773aba0fbee48b100886b90d5a08377cbf5ccf69801ae2c23e1adbf2
EvaluationElement = ae30bab51a34c45a76d00034b29e1c5346fbe3718c386302
8e47226456880a85a2e5118f274a8c260dae62fcec3cde8624405fc7cddbc867
Output = aaf99e5a044bbce915bf3ba381e25da62e4b2cea4cee2f47f3662940284
579c0f8e1e011062ba010ca4f2c67a8157481c9ae7a458ea035a89e1948bfc5b8323
b
```

A.2.2. VOPRF Mode

```
Seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3
KeyInfo = 74657374206b6579
skSm = 167e36766c08353fbc8fc46d89619f0306603a8ed33f7eafcca998ed2e890
d15f64a22b0196aaa298e4a502275d4ced5c6131de64597c500
pkSm = f27f3a898855240ef102d7bd6795aab2fa3972db3d47005cbd33e721cbed5
a3fd37508d093ecc645fa80a7f928c4313cfbd4654e8ea7de8f
```

A.2.2.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = 4bdfc97a75132d92a1da241baff84fada3e7b12d5b712efcac9ba734d54c
2b24bff0ef6310404b5c05d60d7c258cea6500229ee015149f0f
BlindedElement = 1ceb0a3432ac6b583c31fa70b7c17ac86e0aa425e0593d04b58
021670f725eee6664e6cd2041d90f157bc213a2aa4ed7929630b2d9898a76
EvaluationElement = 3afaa02425294a4810766c68e9e4c3c507b109b9064ed56a
148a419371d5fb158f6ab5f0da62a6ba915bbe431097f5c71854821c1f10889d
Proof = f02f7ab2722508e343b5692078556e7ca9b2d63bf83dff902150b867775b
f375693cc6a0adf33178ba7e72d6179b36ed051065c93619752958746f0d52e2e3a9
89d86df15f458847abdcc23976147b7b10c96452332aa03bfce1b89b7aeed080869d
7ce8c7acb7414e7dbfcda298b532
ProofRandomScalar = 54534ad9db9f6df6ce515d1b8017923b65cada199e936a62
3c8eb3bd08e9b3f6584a85e4ff26e9f869d30b6c7c6cc56fd94e306974fbcc3b
Output = b558e37f6435a12fefded196936a4c1d0882bf4a115002920744ecb3128
43678f396f7d36711cf551750388ddf7a53a3aea7fd0ac60568cd2d4ead16a1ee106
f
```

A.2.2.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = beda1edc786e5fd0033feac1992c53a607d516a46251614940e76a2763b8
0683e5b789398710bdbc774d9221dd33c509b4805fc26f0c8d0b
BlindedElement = 2e04b6883057a5b5ba020d077ae36dee76a07c2f3eb8cc55baf
dfb3da9c7405ffe50802f646ca3c3ef39d195c2d88ee56e73825c7cd2319a
EvaluationElement = 6270b2f73738aa846dca34d7b30b7c4f943e31d4d4fb35c5
98f5d608cf25648b44553d43b158dc2707eda170dc439740c10d7b4355bf0f83
Proof = f731f60aa18d508f07dd3b7851fe9f8cfe6f02c4ea2814cfe8af3203e493
44041e6acf0f09fdffdc02d22728544b9bda8d0604e727f27a1efa16526f169191de
db35a1338bf399d8737d6d1638f6d4b895c0869b4194e66fb0dbb4b3e0437a2af0d7
6dd8cfb0bf38c9de605dc5749603
ProofRandomScalar = 00cc800042a0cff31f865698f8858efa75a1f0faef934317
dd6a10bfbbb39f9f2d97dcd5ff4eae02980b08fc68da7b71d39399dc4eb0400a
Output = eb14608be2f14c25b2c9fdd23690d293d0c6aaac501a3405b626b8699cf
34bb9dd4c2d7987b6391519b9480da453611509ba98098b3e79a35acd00f5e9d8abc
e
```

A.2.2.3. Test Vector 3, Batch Size 2

```
Input = 00,5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 89ae863bc6f3e8b59bbd1354548220e81cd0ffb6f9e4ec2173870ae684f8
6b1c06e41ecdb9ef83429e58098b8f30a6b49d414ad5f941cf05,7b1f0b697d9efa2
b60c984df38632f1096f2bf292478118d78e9edabbe9ad22900ad84b1a2cdcb869c7
0c83260691e69ea7f473c3b478707
BlindedElement = c86795bcec21f2b337865406ecbb9495dcacfb0b0d7d2a857dd
31f0f70619a403d42bb57fc53c9182878baa7be06e337c885ba0023190d63,7eba6e
7672c0a7cb7c725ac98adf2b081e05fce49bd5cbc6c0b687aceac45ee0aab63cb13d
0f0493a265996a7aa94c9b30f4fc0c385a36af
EvaluationElement = 14553405aed5bd3b2672fc74f52aa1d9efb9cdd5ec668476
d74c60eca8930994aeaf61eba482173e5953988d702ce5175ab10c1585cbc4a2,727
5d01889988341f8c9c8d0adaedb54af2d166112ac01f2c053fc772cb09d69a33ee0f
fc6a92ca0d752e35f4a33ba0677c37a3618ae07dd
Proof = 820c0da6f0ceb390355da6fb002549f37031e92337bde432d3518541d2f3
e6e4f86fbaa2aa0aa53f15db278a0aa2d305226911e408c25f2bcb3a6774089d075d
3a92e273fbe5359a9c81f9e83082a2e8b02f34d248789f8da583296e7c531e9d8707
90042248e589809a40631feaa914
ProofRandomScalar = 7baaffc0af7cf69078ce1702514d93f32828684a1796b559
988623c12413cf511d13cb07ecb6d54be4962fe28eed7d4386c156301dc2db01
Output = b558e37f6435a12fefded196936a4c1d0882bf4a115002920744ecb3128
43678f396f7d36711cf551750388ddf7a53a3aea7fd0ac60568cd2d4ead16a1ee106
f,eb14608be2f14c25b2c9fdd23690d293d0c6aaac501a3405b626b8699cf34bb9dd
4c2d7987b6391519b9480da453611509ba98098b3e79a35acd00f5e9d8abce
```

A.2.3. POPRF Mode

```
Seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3
KeyInfo = 74657374206b6579
skSm = f68691e40ba92bfdf37acfff161f5404f9ae0e53c7cedb0a790ab17c4c0a7
4a314c24974057464185e2d2e648f74ee6663443646db2c111a
pkSm = 2a742a63231b139ce19eab43e7a855f32e5dcbd16ef52a7f968456a814104
5d49e3e28a995cfaa22ee104e22f2239f624b3fa7d41bf15186
```

A.2.3.1. Test Vector 1, Batch Size 1

```
Input = 00
Info = 7465737420696e666f
Blind = ee671e4c9b6783bd5e4a55d2e8474fe0ec811b4cca7c0e51a886c4343d83
c4e5228b87399f1dbf033ee131fe52bae62a0cb27eb7abfcab24
BlindedElement = 4c371528ab436b8a6a5bea333cc5702c70cdddb80d12dc2eafa
06b87c15bba8b0b5451bc09f3d07e57c12af4c0398b09ae91b678fdeaf2aa
EvaluationElement = d27f65d6c41880303989752e40748e940add1ad32e7f76cc
bb873b7fff424d348ec8e43c11402e02934c1fcdadeacbca2d2e5171daaeef90
Proof = bf2f61413c56c0351151c1995007ceb2e197c987056f20a54f0027e544a0
b20a7891b9aa882203f2e09e1a0ca9464e3cdf130eea9e1123023460d3f280dac87d
23b8d2258666d002f57810d8847832b775984819e457c7bbe703947e7aeccdf59d3e
520437edefc26b814f9fa7fa9917
ProofRandomScalar = c4b297c662a87631531aade91c0558d87224d92247bdfa41
9a53af4cbdb352b0a2016e5e5f6c0bee4a642526ef9910289315b71fdee5df1e
Output = 1ffbf9591b674e6a089279a8319c75e949cc277d7b5c757361412180307
90755e90af009768e1b9240c9734d8886c6121123384140b26c38c7a6c4217a1b3d9
4
```

A.2.3.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Info = 7465737420696e666f
Blind = 1abe4937f28f531b14ac96b844320e7a66810c2d9391cbb877348301ab59
a3a91b4a2129672886ae5da7839f2ac8cf1c5fa92703f5b3fd06
BlindedElement = dea615b00285247715173fc6db40cab1436607bc0eaed3d7a1a
1467b70c7ff2f2ce91c05bcaeda2b01952926f254f13e1a763a174caa693a
EvaluationElement = a692017b9e91efbe6641c3ef0cd3b352022ed08bb5ed0c1d
a0838589bebe53c2d2959818359cb0213b94cafa672673608b9a2280671d6c75
Proof = ede122b8ce87d22fa1bb9dd38dc76da1a9ff812a8d2cbf3d2a6e86a10331
a849d203bd925d6f130d80f333aa0443488731769e975b4c900d923d740fec13a61d
3175a0daf9a88d8f66704b36ca2b1b7fefd6cab4ffb50fe998e53ce4743ee9466a56
886f79fd6d5b924553f64130c60a
ProofRandomScalar = a3e896e126d371f6380ca41757f6458b93b049e1b0d73ab5
b8d914b08dff3e52e62ea8898d35b2862d28ff4c5f89353d25d6b5a8dc014d3b
Output = daeb206a0e1fc120ebe4ad885f851f456f7d8908166839b7dc541f71251
4203d9a3589025b4bfad6a79c6d40bfbf217f44a9aa17874a1ec271b23cced72a44e
f
```

A.2.3.3. Test Vector 3, Batch Size 2

```
Input = 00,5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Info = 7465737420696e666f
Blind = 255d8adc40b8f39f14cd8bd4ade8abbb95166afdc9e922203abe7a853985
4c64b943b0b46e1e1b47cfb52e9a0867c8cde22bcbdd724d9f09,71bd897c56c86b3
1b096103b7e2d26d0f4d66be95299379b41668dbbc5ece26cc212d9f2cbfaf479efa
17b7f6b056dfcfbee5bd7365cea26
BlindedElement = 361b80bba04ff4b211e38e636a8530531213a44f44738992b18
eaf0d9759eedcb7e4034e9bda6f8829250aff72343b0d2d1e23d612d94674,c68d79
d1a614b90e6ab1dc14a982f9fdd423edc94a10d87d45e32935e363079967ad289482
8b1764cca8dec5e9f919def474b1d03b6c069d
EvaluationElement = 32629ccfd36787d8d80756f025f6c23c21145dd22c28d974
f34098e166a300731b691e1faa7e3959c1bb38312c43d1d693cbab4b90fe7d2e,eec
655a1a869b3f0f470f7a0f2cef69eb6539c6c1b9e49d9b380bccc7b510d466f45d88
fa690b687a8507d1e0b275028d095292fc4aadd2d
Proof = 2f3501925c81837232ae34e5351518ad35e24f1d32f7459da3c19cae7746
95e7dc1eca32133dd57cd0e2eb67c75c9edd9cd3ff9c5e1759314ea99a4eca322f6e
56f4b80795f67d1bf747834d2d7b3049351979ca876ecf28f87b81fba243269e3c09
ea1889abd968af67c7ca511d0c3d
ProofRandomScalar = bbbf1ebe98b192e93cedceb9c0164e95b891bd8bc81721b8
ea31835d6f9687a36c94592ab76579f42ce1be6961f0700496e71df8c17ab50c
Output = 1ffbf9591b674e6a089279a8319c75e949cc277d7b5c757361412180307
90755e90af009768e1b9240c9734d8886c6121123384140b26c38c7a6c4217a1b3d9
4,daeb206a0e1fc120ebe4ad885f851f456f7d8908166839b7dc541f712514203d9a
3589025b4bfad6a79c6d40bfbf217f44a9aa17874a1ec271b23cced72a44ef
```

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

A.3.1. OPRF Mode

```
Seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3
KeyInfo = 74657374206b6579
skSm = 88a91851d93ab3e4f2636babc60d6ce9d1aee2b86dece13fa8590d955a08d
987
```

A.3.1.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = f70cf205f782fa11a0d61b2f5a8a2a1143368327f3077c68a1545e9aafbb
a6aa
BlindedElement = 0372ffe1ebd9273f17b09916d31e7884707e8902f7e3af2a1b3
ae1dfbfae9b5126
EvaluationElement = 02aa5b346b0375cd734014ffa9ed2135a1b07565c44fe64d
5accfe6ab6d8c37f77
Output = 413c5d45657ce515914232ef0bafdbc1bfa5c272d4b403f2cea0ccf7ca1
8f9be
```

A.3.1.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 482562df55c99bf9591cb0eab2a72d044c05ca2cc2ef9b609a38546f74b6
d689
BlindedElement = 02fefe6e044601a158175fb4bf90c06841ca7211dde4e56e5ca
c6dd45728cfa04a
EvaluationElement = 03167ed445f79ffa867268e30c0aa240ad1a863569016406
6d833e350802e57273
Output = 2a44e98a9df03b79dc27c178d96cfa69ba995159fe6a7b6013c7205f9ba
57038
```

A.3.2. VOPRF Mode

```
Seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3
3a3
KeyInfo = 74657374206b6579
skSm = c8a626b52be02b06e9cdb1a05490392938642a30b1451b0cd1be1d3612b33
6b5
pkSm = 0201d3da874a209120ac442081e9ef9ed8ee76fda919d0f386cb5a0143755
b10df
```

A.3.2.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = e74c5078a81806f74dd65065273c5bd886c7f87ff8c5f39f90320718eff7
47e3
BlindedElement = 029d750421c5c726658902c47d3675ebba01ba25d0bd127bf6e
338b801b166f1d2
EvaluationElement = 0291e9890c7418a2fc1ac635d2650bae3f1a25a9ffcd0bc0
1b3c39fcee4b095dca
Proof = 54ec2d8558f5c72ff32489556c3ba1f3087810c5f51cc025f07adc034df2
dcd6d706e7bdae3119b70748cbf76b66d520de87bf90287a091cf6f8d2a465cf2200
ProofRandomScalar = dfc19eb96faba6382ec845097904db87240b9dd47b1e487e
c625f11a7ba2cc3e
Output = a906579bce2c9123e5a105d4bdbcafb513d7d764e4f0937bee95b362527
78424
```

A.3.2.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = dfe89ac0cdd6b74684580de0f95f8e06aa6f6663d48b1a4b998a539380ed
73cb
BlindedElement = 02e6085d4017ae0bede4b261977b588349d323414eb5c409e55
2e2bd4c82df498b
EvaluationElement = 03a649c5ac48f33a6c6cd82120145e673e17395ca94ea824
c7d2dda7203ba4159a
Proof = 1a79f6a52579f7acb0100c916390989a1dca3c1b3078402e102b8dd037f0
b34d929d38239b34175f1328708ec197bfc532ef31dafdf1ee85db4ccf8769844fdb
ProofRandomScalar = 4f9a70536c175f11a827452672b60d4e9f89eba281046e28
39dd2c7a98309b07
Output = d13c62d285a71acb534dcebdf312bfec0e2a3fcb79f4ac32d2dfb0bc9aa
e3cc7
```

A.3.2.3. Test Vector 3, Batch Size 2

```
Input = 00,5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 9e68a597db2c14728fade716a6a82d600444b26de335ba38cf092d80c7cf
2cb6,d3d6e1e1006dc8984ad28a4c93ecfc36fc2171046b3c4284855cfa2434ed98d
c
BlindedElement = 03b2332e9dc41bd9b7997df58c1f432d13c4f018cb2095ef8eb
14ef3b323aceb86,03eea6961f7f16deaa8deb6f68a865ae04d8be760626cad589b2
2cb90262e30b0b
EvaluationElement = 021f7b60c7c53fd3a6867cb38bb7f6febdbd802a78d10111
00a779b67a801c3bbe,0266e83c8c525cd612669737496f0a736feb7d4209a520d2c
dc204971215db0262
Proof = 92c5c32bc18f3f5dbdd51473f4e3ecc9b07797c63d679be5399b223ae801
ddd1e469df512f907d317a0930dd0e644b26c96edc87d2f8e0a09e66bc73db8647c5
ProofRandomScalar = 6e953a630772f68b53baade9962d164565d8c0e3a1ba1a33
7759061965a423da
Output = a906579bce2c9123e5a105d4bdbcafb513d7d764e4f0937bee95b362527
78424,d13c62d285a71acb534dcebdf312bfec0e2a3fcb79f4ac32d2dfb0bc9aae3c
c7
```

A.3.3. POPRF Mode

```
Seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3
KeyInfo = 74657374206b6579
skSm = b75567bfc40aaaf7735c35c6ad5d55a725c9d42ac66df2e1dbd2027bde289
264
pkSm = 02eca084e8d6ac9ed1c5962e004e95e7c68a81e04be93ceabf79c619de2bc
c3eb9
```

A.3.3.1. Test Vector 1, Batch Size 1

```
Input = 00
Info = 7465737420696e666f
Blind = 4238835743037876080d2e3e27bc3ce7b5fb6a1107ffedeaedb371767432
b68c
BlindedElement = 02cb57f07ba100b93ce1bf8176963c8c7f73a76827f1c1401a9
23d7ca4083e15aa
EvaluationElement = 03059d58ec9a801e33f57525c03241d8ffb61b67a18edd35
222d864ffbb42b5d2f
Proof = 13889d6849850ccd0119981fc053a38a30a57d275091df2887943d1332f7
38204f8a6cf2fb6e57c9b118ec82b9b012f8864561e4cd8866245f9c762b9d45dbf9
ProofRandomScalar = 3d5c65b55a1b8960563b3420d7764097502850c445ccd86e
2d20d7e4ec77617b
Output = 15fce9922a2307349aac2eccc41941283e3c5e938aaf2506f99a6d8b6ee
34ef8
```

A.3.3.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Info = 7465737420696e666f
Blind = c262bf51dc970d63acb5ab74318e54223c759e9747f59c0d4ecbc0873026
67fb
BlindedElement = 02208809631cc08f553d7843db566c55746e760a77c63513d2b
22096f98452cf9d
EvaluationElement = 02301ee1cf1d01276649ac0f718ebbfa1c0d6a1b3e7ea82b
3085e9173910fcb0ef
Proof = cb69a1ec76643a2100cc9bfe6cf1ed1fa5ba3612ed3e3211036b5ed835a1
38be3eb92126694e3e925ab138d4df885be18ed80371847f80baab82ce70588eebaf
ProofRandomScalar = 6c6990f0fcd9a655f77ff0b2ebcfe21e1a1ca4a84361e9f1
b18e24c9a40ed5ef
Output = a06ed7380210856caaba173bcad06266186c6638d86e372c3c96b9bd2f3
53543
```

A.3.3.3. Test Vector 3, Batch Size 2

```
Input = 00,5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Info = 7465737420696e666f
Blind = 3f95c9b1334d8af16ae1e69f5adc24e5aa89ebb63637c835fd39b17a1a44
53ec,9801a9d83b5d1c0fc0812c10e18f146b14d7eb94755a918bac1ef8d69d21a7c
2
BlindedElement = 0277b1e97f06cb0976bf196a30dc7b3f635a40ec1337d4bda00
f809eaf244f7133,0306823c1610cc81b08db444d7c23cd368e8b6fc1a7fa3d727f2
8bcc7d3afa9c41
EvaluationElement = 030f395a0e0328018c78de95ff498a0afb54fbcf34197226
49e211940852ad0171,0221335509649461a4d201d2887af62313466af660559c348
e8ac326fbc1c147af
Proof = 187aaa12108c49c1395001ccaa677519572ac4680b0f41b346b9879d3ea6
fdb9fced5c7f20b351d03786d031de79cd3c03723ce48053a13b640fe6051ee3584e
ProofRandomScalar = fa0ea4754fb56527be010296ea880e1c6a4dbbc9ede543a2
ad0f83fd60fdacb6
Output = 15fce9922a2307349aac2eccc41941283e3c5e938aaf2506f99a6d8b6ee
34ef8,a06ed7380210856caaba173bcad06266186c6638d86e372c3c96b9bd2f3535
43
```

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

A.4.1. OPRF Mode

```
Seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3
KeyInfo = 74657374206b6579
skSm = 8b0972b97a0339dbcdb993113426ce1fe1b11efefe53e010bc0ea279dda2e
37ac7a5599acec1a77f43a3ac7a8252782f
```

A.4.1.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = cda63dff3137c959747ec1d27852fce42d79fc710159f349e7da18455479
e27473269d2926fec54d4567adabd7951ad6
BlindedElement = 020db1b05b22ddd8a851792dfb5b10b4f237d69522097cbb012
7ae537e3256f86e35a72554a6ebdb26c28342fee16473dd
EvaluationElement = 031e6e8c82d3284727a724a5854b3e2bf9958b4e5470601f
4ca37d33d26879eca817796cb7e98bbbb1d1739eeafb33c027
Output = b2e380ca96ea80f7550a6b663e5f7752d7d7772c46169d72308a8425903
1e804ba577ac34e632f535a9519a692734016
```

A.4.1.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = f9e066cf04a050c4fd762bff10c1b9bd5d37afc6f3644f8545b9a09a6d7a
3073b3c9b3d78588213957ea3a5dfd0f1fe4
BlindedElement = 023c36bf6352c93d27b118972d1040cf22f99d5a1c8134afb89
8d30b319f70a096973db23410881f84eea599c0c73220bd
EvaluationElement = 0240b6a002d0190793ea62a7499244027753d63b0a57cea1
98c8c6dc883cdeb273ab385699bb414f1040bb6819313cd675
Output = 1d155a7ba2ea75c4f1e76fb0a37231e9b0776eed3f24a6541a01907ca8a
fb984a74408e6d2de8e481cae5dd03bdae3ce
```

A.4.2. VOPRF Mode

```
Seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3
KeyInfo = 74657374206b6579
skSm = 70855cb96c961b39ea3ea5776d89c8b7623f5891a26e8437f86e2c713bdb0
da23415590a28184dc22088a215ebc7fe45
pkSm = 02d7bdae4b97ecf0fbb8c00cff3a3a9b6d0fb0cc34f8490a98a74dbb59a85
f43bda8ca7b3c0b05164f38d8efdef2c3426a
```

A.4.2.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = 61247a74d0c62c98ddff1365bb9b82b279e775b7220c673c782e351691be
a8206a6b6856c044df390ab5683964fc7aac
BlindedElement = 026601d99c313b827a09aad832fcc814ac5257a57bb49d65c05
e247df9518315a66557fc8af56b4521c51900aaab1a2ea9
EvaluationElement = 020b478b9c9b1a5935e07fb532eac2e596b78170a0e755ec
c71829419e63a2119eae23be281e109de205cd85af7e42228a
Proof = 02d0946f1795048bc803171aea5b4a9a5f256bd5fa9414e5fa76dd17a4aa
a94307814d57c2cca239485e29bb76d4ac1b4d3d62dfbb8e43c7135b2ebe50fe923e
30bd99e1e6ec961db18fa6e67c63dd6652284c15860156c08d64d838efbeeb68
ProofRandomScalar = f5685928c72d9dab8ddfe45de734ce0d4ff5823d2e40c4fc
f880e9a8272b46eea593b1095e7d38ba6ff37c42b3c48598
Output = f18884ace2e342f849cea7f2f17de902b9884574fdaa8f507356f482c6b
67013f329e8c899b3c2c154af1defaa11d656
```

A.4.2.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = ef54a703503046d8272eaea47cfa963b696f07af04cbc6545ca16de56540
574e2bc92534ac475d6a3649f3e9cdf20a7f
BlindedElement = 0279e61686e698fdbac5cf484f54846db8cdf6f403fa88209c3
4c56c584fe4ca600ac81b61aad11c5e639ff1add3b30de4
EvaluationElement = 03900e8e3f5b8bf698e7aa0aacb8dbdfaedf80220b1f640d
e2049615985b19b913569cc2feb90725a3661146fb88ef3755
Proof = 343536346a1145b81336eafc239f225dc6a154752492707c0465f029aa9a
f0fe2bf0428285e43b596db633b50f0801b62b0e9c64c62f329b8a84324a415e4a58
6cbf9477b1285c9b74f614c352e06658a8997486b8177006491e84aa96a3de09
ProofRandomScalar = 0cdd9475ad6d9e630235ff21b634bc650bf837aaa273530d
c66aa53bb9adb4f0ed499871eb81ae8c1af769a56d4fc42b
Output = f91d172cdecdea4f8299c8b39426db4c47428b82f8872b8539ad9b019de
b48b8d3c928c572ed988d5591a4442c060438
```

A.4.2.3. Test Vector 3, Batch Size 2

```
Input = 00,5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 485cccf5018abbf875b8e81c5ade0def4fe6fa8dfc15388367a60f23616c
d1468dae601875f7dd570624d0ae9d7be2e7,b0d53a6f8da29c3cf4f8695135d6454
24c747bec642bc91375ff142da4687426b0b4f35c14eb2477c52e1ffe177f193b
BlindedElement = 03dbcb21b211e7b5d2cf0c36d782308af28458539423f67a293
36355e55035137eb768b1935b5a825c589a2913f0c2894e,036c8b2fa4dd9cd05756
1d377b4686cddc82317ad3e5eda08bece2a8616ca724937ff933e340a47fc09bfe9b
0fc1ef9ab6
EvaluationElement = 03a1cba477a408162aacdca43e059309fd61cc14687a107b
d492a1ec688a010ff49c60684e0f973412a7da2e627b1553a5,036fe6df8a99bb7b2
c4a5020ab4c6d7e71b5abca2d5d5a418f2314b614deb40c7b3acad982951b5f524e5
6f0e9ac7d8e95
Proof = 1e26bf1210717b88dfae585008100e9ccaebe93b8605ca168a608cbf1855
697b7b87d0b9c6bdca85e43143b3630e87f2fe9ce519dba3d477d2a869bcad0db9dc
6239cd11938213f9bfd63d39de090a6fc90cd1f33f164b2c54c38bc31ad98ddf
ProofRandomScalar = b36f4c2a140b7a3c53dd8efb6171d3bb4d73591be8483a1a
38e40c13a04b0f2180dda3c36e3d43c3a8f127158d010945
Output = f18884ace2e342f849cea7f2f17de902b9884574fdaa8f507356f482c6b
67013f329e8c899b3c2c154af1defaa11d656,f91d172cdecdea4f8299c8b39426db
4c47428b82f8872b8539ad9b019deb48b8d3c928c572ed988d5591a4442c060438
```

A.4.3. POPRF Mode

```
Seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3
3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3
KeyInfo = 74657374206b6579
skSm = 2b65a799c107905abcc4f94bdc4756e8641112ae1fc21708cd9d62a952629
38ded6834e46bad252b4e533ee7eec7e26e
pkSm = 0286f37b6295bba7ebf35d2bfbb944d441fc416e51eb5ceeb63ac98afa6a6
27ccafe20bd600c728bc5b1300148ef2ba6e6
```

A.4.3.1. Test Vector 1, Batch Size 1

```
Input = 00
Info = 7465737420696e666f
Blind = 9572d3a8a106f875023c9722b2de94efaa02c8e46a9e48f3e2ee00241f9a
75f3f7493200a8a605644334de4987fb60da
BlindedElement = 0252f98f04a956afa469c62ca2850f751b112dc019d4e713c66
2fc0735ef8573f1497cea55b750f27f0efc8330e394a3ab
EvaluationElement = 03fb20c33a7f6f01f2bb388318a6db84f7183bc3bd5e5840
302fe38b6b313649b523238b4c4c625614440dd6ddbbcc7272
Proof = d33c83c1840a48759659a4d417769ae3bb1adb86326a36fa1ff24f70066b
75d0200e5c1e7d9847e91f7d3d6843efc62101c401a7c952cde32ada6fec848450d8
564e2c778af47ece4f50a88c6d2281bdd858b90fdfad8b093c986bc1e59aaa2e
ProofRandomScalar = 7e82569cb56d97e9c20e59311bac3a50735d573abb787b25
1879b77de4df554c91e25e117919a9db2af19b32ce0d501d
Output = af52cf184180177970be0770e1c7920aa307b767556a13de38a64723d8d
cc7b344af9b6dd8f117ac2cef249ee3acc8fb
```

A.4.3.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Info = 7465737420696e666f
Blind = 01e6e57b7ec6752a45c74f3ed36a3eb8ad0dafb634f668e415357a04fab5
01c0f6764e854701129e38071b008286c5fc
BlindedElement = 0257c264a1016e7a1a8236e46cb3bc11a0f13178b03262e0153
1da14a05e75a811ba4669fc41cc9453298f71c23834f91c
EvaluationElement = 0295529cd99f4255be59966e430bef38c93a5261b0624612
091327c9aedaaaa40d22b03280ec15620bc91d48970f18c68f
Proof = b5dfd5fc5ddc61ad8234c544aacbf280193da985d9204d5a30ef9d1a5964
c1e70ffc3d9c986c93f561ab6f91f012c8ef9b9b6f2d1c178f01fe172c37c98fd4ef
05e5b15c15e810241ed6dda051500165d9f79d4e83580ea4c810a95dddcb593c
ProofRandomScalar = 6b61028c0ce57aa6729d935ef02e2dd607cb7efcf4ae3bba
c5ec43774e65a9980f648a5af772f5e7337fbeefbee276ca
Output = 8bc546462de3087cddafcf81435d5802c0c31f557c791b115a092d5b71e
a2b6e20986bb624ead85c7a63c976c05dcddd
```

A.4.3.3. Test Vector 3, Batch Size 2

```
Input = 00,5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Info = 7465737420696e666f
Blind = ebd2fec41edafcba833ccaac567c14d2fa01f55b33a2fbbb37118f2f5603
b1298346e02cbdf55c95ef9b1aadda5ef281,be210603388cbcabb8cb630aa1ad04d
73e349009a438ce248380bd4b7e6758211fe9692922fb61f00f1a39bc735cefce
BlindedElement = 02b1938a7613e9567a67aac83c50529238e3323c212dd407491
1980c4d998479174ddb9c925d1b761b33da2ea0bd0ea057,031b9a47b7caf732ff32
db035d1d073fd925c17dbb6c83e00a49af674166bf264bdb00c303edb26af96fed6f
e9ce44dc36
EvaluationElement = 02c75f2d383c18692e0e11b08e9187c4c047d28116977c8e
5e1e872f1cf5eab457c04fd50274cd5cc4b1996a607470694e,02e72fbfff4c70479
23b967ee9a6b37d902b49a465242c12b2b910daa5f30c3f947899283ed0a6c758348
55a1ac0c64065
Proof = bded438b699d3bb8bab26954f9a7fb5bb402f043c3364dc4f2b68976748a
77868dec1fa2d0774d306043ee8abbad5ef8ceee6a331be2906124f53f37c96d7f5a
4aa543053ccca87b577a32d803c3ca4841e37b3c4b5cf20aad11c59dec72a350
ProofRandomScalar = c7a86f11c143a291e349b70b34e67b38fe9dc6f90b473750
87d72e891df74070810500dfd391282c15d87bacdc9867a5
Output = af52cf184180177970be0770e1c7920aa307b767556a13de38a64723d8d
cc7b344af9b6dd8f117ac2cef249ee3acc8fb,8bc546462de3087cddafcf81435d58
02c0c31f557c791b115a092d5b71ea2b6e20986bb624ead85c7a63c976c05dcddd
```

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

A.5.1. OPRF Mode

```
Seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3
KeyInfo = 74657374206b6579
skSm = 00dc7a8db919a1076810a0c1503716d91668fa9edc60952317f26d47a090b
70dcfd3f530d07f48675cf8236d1daa81f3ff0f289942632e5cefd27a2190f0cefdc
302
```

A.5.1.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = 00b638b3000884019316267eae9b424f812592e4dc9cd7f7aebfb1d3d2b8
c7fa7904503aef20c694a01d3e1154fe98e7232be9eaec5789a012a559367b1f9965
4ddf
BlindedElement = 02016f3fc7b3c84f673c75b3bb3e00ddd81e734cc84fe3bd4a7
671e0a971879b7678c048f40aae87179614abc2261522303257a92127a195298744c
54094b7b87499c0
EvaluationElement = 0301ddff1ac88acd812a2917cee4917f8a692eaabf9fd052
9981441b83e368175b566657729a8be5ba2573e33e7734c146ef4c8b7d41f4503842
80797318ff3a62d79c
Output = 383e3098d74b43f75d2e1136d7e7c08702d992e6f5f24f2bd438f98b86d
9d143ce87281b2daf7d67c94370903ba81495655d6e9626443a895b37bb74c0276f2
a
```

A.5.1.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 00219598d5f1544830f9d667b683234c68ef3db95227fe3ebdfd963d0307
0055fef107bfeb3c79c86b934061f894227b23a69eb0b53f168a4a2230ef6a7d703a
c4ce
BlindedElement = 0201e75b88d5de2b839f356a05c359ed610601450d83dcc9def
649fbd00a9790161e9333cb07978d1567ecab037c498ae2b00d9181abfd7bcee3fee
dc11de88c54190f
EvaluationElement = 0200b99d9694bbf9b9ea783e18e5fd049fb2fdf169cf386a
be304ccf8cdd633f7a1e25083ec6a6ca3a6e82367b38ee3c991e024097cf6fad928b
023817cdc5dea21751
Output = 5100f12a88477ba993cfe8eb5a82a835892b7fa3bdb47dc1db19725e4c1
138798e0f965df4f649e3a159aaca1fdd07034f7b91c0c9ac3d064b50953bb5c867c
3
```

A.5.2. VOPRF Mode

```
Seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3
KeyInfo = 74657374206b6579
skSm = 00ef23ce13076b43a5a33e8fb8f94c940bbeabb762e5380d69cc9fa82c22a
8de39431c99e8b9d9fb75ea90446f895db04fe402b8be2c9df839f7d10ea6a23e7e0
eb4
pkSm = 020006090cc9a6f2eaef7e12759a8b5362e9972b4f36c4b3a3d71c4b67469
638593ed8f46291542e0f04fd462a8e8ab96047be087a9d3fb182f4c138c6fc95659
3b205
```

A.5.2.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = 01dd6b45efbc57c5f087181c9f03d5b5e51b3a90cc9da17604b2e59a9375
9eb0985d2259c20e3783be009527adf47f8fb5b1437cba7731c34ac70bc6ab1b8c14
ff42
BlindedElement = 0201b93fc5997dc0e8acd5b3ffa3a6f1be1522986c17ed60e5b
ad7b057136b3b7e31b5a7073a744a1304bf9bce4a27b02d77f1caf73a5f72686fa9e
83dbe9f730b4304
EvaluationElement = 0301142205a8fb983efb6d76111db30e2a6e7c54724fe0ee
54f842a477cbf03adcb2cca8df2f165a65694e7a056948f8afd651b32ea8153cc26f
819cc5b1243f383910
Proof = 00fd70ea3e4b7b008de749adb7403ca66f7aa56ae06a587d7d74a06daf6d
5c4dcce8a81acdbee1fcd21ce55db4440383b9fcfc1d584db6fa022a5fd62595c7d9
39820116ad656f5ad470b5bdd208248a0a0b064960c80e180239691c5eb77b4a760e
eda8c27cf3cddc37d6329ad4997a37ffe9aba6f96d45e3e67fad86ec7b1d3a47487f
ProofRandomScalar = 01ce330164821b9b2a108e3ef8964622075015ac9ea0f838
0dcce04b4c70b85f82bd8d1806c3f85daa0e690689a7ed6faa65712283a076c4eaee
988dcf39d6775f40
Output = b3e837431aaafdfa8efbf486d70ca2d4364ef86afc7a8941d9bf1a6adb7
bfd8c5302f91ee5796d956b5d3ea95fd0138d55d3059b1f4febf8cfd552e31fa2cf9
7
```

A.5.2.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 001745a97be4680b39889979a8b4b4322450628389ff2d90c0799597e99c
926ae54b2fce5ca13daa8cabbd4da53324fbd20554f2c56460442edb7d6ee76b64ab
68d1
BlindedElement = 0300c2eed082810750dc327122ac1d9de647d1943f1767bd546
56835ef71dd68347436c121df49c997b0adb11cc421ec8abc5611b6d9e86468e4666
001e3db386ace9f
EvaluationElement = 02002ddf4ceaed6eefea8f12ca765c6d800e3f514f4fc75f
52f55a555cbecfb724f2bde39d5890f4b0dfcd3ea9029f663c7babddb30dd0692bfe
0f76190b7ee30bf68a
Proof = 018d1fbf1bc79cb4636d905f03bba48e0d872ac89c4c7e8bcc6884dc796d
22ceb648dd373b196d23335052a8d8013b154b38d233d68612213e3cea8f9024a1d0
1e160004438ef7171638b7799a987739064e2313f9a902413d1f11f0eb5c7e8ca00d
9c83d03aa12bef8059144bc169bec7f4258858438daaa3ebdacad658e5f3eb7c6d5b
ProofRandomScalar = 0013559a0599ff077b4ebcbe7f73e9fc1bc25fff3fc5fd6c
8bc664e27822fdece106def4a69460e9777347a314fbbe5035803d3aa65819e81997
c4d89909e25ce20e
Output = e8f92bac6c7ae89918d724697d8c45da339f55b61d527c50104e6658280
3a8e6dcceae31b0d499e471aca460194a011d6b8b94fe2886b8b5a0c242079bfbf09
c
```

A.5.2.3. Test Vector 3, Batch Size 2

```
Input = 00,5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 017b1679ed98960e4cee27f330d5d3dccebf40596dc7e8b057938841423f
8b336f12c6c4dfa3a822d8f670e5aa46e733baaec9f93d5e14ad9ab99dfcbcb2ad15
7a8b,00010fddc6356f1aa3fb05702631e213b4bbbe8fe5176fff25526ed5b1772ba
6164952c3c2da8017fdf337f81f5cbd0ec805923a335fa1bde3dbb840b3924c5ceba
6
BlindedElement = 03001658908fb353f15fd637ec5b9703cc1dfe5a8aab4f5fd51
9c0e41f69300769918d28963c07e5678ecb98c235c406f29dd1cd1dadadb4e23b98a
1cb290992b9aa46,030123068d8554e65999a6df1fde0bf10550ec5e223ef1d30c07
3dee509933502a59106aae80e67d33f93400e2c32b5d9bd49f4e8cb97f08f4181998
d330013b9e07a3
EvaluationElement = 03012d790d77da1675a8e4ca5d9ce622046e31ea4af9511f
a32ef80fbe60e9e04866feeb7f31818c1ff0ed263cecf07f787129450f9f7322fa9b
593edc5cb626d1abb8,020198f45b9988bf0ab56c37bbaed0df4099c9b7b7a89b5c7
030a451687cd9f8fd777f587dd7dcd717f529149d21c0763da4a03441d96e6fa2458
88a352d08ddda70db
Proof = 01d6f0f91660fc391573f819587ad310bf3da80a2dd4cda113c1fbbec9d1
cd88a0e4d783833c4c64bbcb6414688d925af2a4a5845b2e3b6634df5e5d901047d5
148f01a1ec61207132162fc851126b07397beeeae6814e8e135e4a03ca8bd346568d
23c85e0a64b9b83de207ff8d17674863bde02eaeb0fa16d05a9b44cb01654bd0806d
ProofRandomScalar = 001caeef2365ebf9c1edbdb24825e5735614aaf644f03458
a1f30c90229f8068bec0ae930eef110e98ea1cbc6d849b4c9ca5b7a970d0320ba5f4
f95f5cd4f501d720
Output = b3e837431aaafdfa8efbf486d70ca2d4364ef86afc7a8941d9bf1a6adb7
bfd8c5302f91ee5796d956b5d3ea95fd0138d55d3059b1f4febf8cfd552e31fa2cf9
7,e8f92bac6c7ae89918d724697d8c45da339f55b61d527c50104e66582803a8e6dc
ceae31b0d499e471aca460194a011d6b8b94fe2886b8b5a0c242079bfbf09c
```

A.5.3. POPRF Mode

```
Seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3
KeyInfo = 74657374206b6579
skSm = 01f8c6fb8fb3265b70e97870f942f7de645132f694c139446ab758871bdf0
058a2a83b4679fc9fc1c6a0936b2f2e8e079a75b20688d4fe828e74d16bfc6255289
92e
pkSm = 03013db33ba3e475e5696be39d99fd9ffd96452c4fe78df4eef5723097943
1f734aceefad464c4885b99313a775f5f4524db3c8404400169fd139ca053b75c6d7
e848e
```

A.5.3.1. Test Vector 1, Batch Size 1

```
Input = 00
Info = 7465737420696e666f
Blind = 00dc9f04fb076cffe7d179d692a05b0c2210b6c008c1062c1e54514ef654
eefc0519dd1867571c9d518e305fdf463231b6ec8b7498e2122a7a6033b6261a1696
a773
BlindedElement = 03009a6b363627cbc6ba5f241493a724a69ca7a85f203fb5100
bde9f36ee57e3fe75a5b41d10c6d9a2799fcee9cd1f4bcd730cb8d9be7aa5e8a7a48
8b6ae3004afd2a8
EvaluationElement = 03009ae81470679a5c5733401488cc6648a522a208e698e9
879307e794158ce508e08a50556ec66a055f05f5d5276231258d95d004a49a308037
2f3e9d2075753c010f
Proof = 0122e18e5c3e2242617098cf1d6b5868d66fb4f4816ddd3769e5b7f326f0
ea3d79cd8b8b87be31c1acb9559a2ffdd13f4af7ee143e5081a2db996f3a7d2da839
73e100f559c9dbb7b16df3d5f609d2f8f2184e9e204e6444db72608e4816beee31c8
59dfabfe137bc3bae06947d767cd8cb6ad634134cf6faec24bc8341d51b584872ae1
ProofRandomScalar = 00c07a53a1c70f44466b3861be4f8ef48c2bb1aec2e478e3
41c467fd4a2638aeca63ed6c4bc48d008bca3f36f043e0eb73a44aba77e5e37d5ab1
389e09b80a34cfaa
Output = 70ad5e29de9f6e35f16afab3b97c1b26fdf6be0da60aff48a99980ddb8d
7c2d728a8a5d2837179bfddd612712e014c0c9b9596cbb5a6ee6761c564dbb8921b4
e
```

A.5.3.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Info = 7465737420696e666f
Blind = 0085ad3fc8c91caec3bd7699591b10d6da93877a470e128f38030627dffc
bbf1f576b38677841fc47af778f9d85ac9bce6279388ddf4607e295e64cea6f4f950
78b8
BlindedElement = 0201eeaedeb3692cc0ecfeacdf9cab61947eb0d23bbfe2e1fbf
8de0907f9410b6089d060d3af63411fd81b9d588fa2c48bf8ec63ec66c14b86d2371
24042ca83fc99e1
EvaluationElement = 0300886138e19945036ebe6f4195cf9f688d9e5a7c89597d
feea6e0e5fcf4b53a9dfa280c8409b6abe8051e3394279d0b669440af8a27aad169d
e10446eb88e09d6801
Proof = 01cb4d8a14eeb472ee3e2fbfe3f6d49f3654cfe6238254bea17ce30848ca
934e20e82c2a33d140de55b24fab047811e20b46f6dcaf3c0945c802e84891316186
17ef001e233aa2c3d674bce7465278faab6300d4f6b5463e1597d74e2a69865bb068
1604f9210edbf50bf341d836dc09af85e603b4b2b8b55c90c2efd979a4e312b653e1
ProofRandomScalar = 003a09eed29f2e7f8950d766270d390db7a53b8080b89cb9
e024e1e008d83bd90e94f501281b6b49c351c959348b3a65f24c6f74e77a62905a6d
3e4b0b10600a7cbc
Output = ee2d8e42030da6283ab59a11f41a171c65e208306e00c6f965a56c10f33
bf0942bb38b7e1a33c70bc3542d27220379cbcef8b91898c720be948e9db214a14bb
9
```

A.5.3.3. Test Vector 3, Batch Size 2

```
Input = 00,5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Info = 7465737420696e666f
Blind = 019dd87ebabccec2627d4006b698d9ba57f6e207c989448d39fe0431e60c
9a9a4110596d5a16fa6cdf3f66467524f295b5dc8f3492c6da02dd7387bd1dc40065
b232,00adaeeed48a6f9a8fb57640c3bff88d3ab3cc52ef969f02beaba2c6e32c2f3
7baaf4ee9c691833dc081e2a0fb6ff636525457a21c1fc56bf3514635ac7fb8618f7
3
BlindedElement = 030170a994d40e8517aed3a7efae01b650dc131c1ae07158f02
ad70d211348a4b328add9d17e93d2e747dd8bc6960a5ab3bec6a9a29f793bf5663d0
ab108b5f84fa751,03011045c5ea9567626cf6d2baa158830b035e66c249df4967bb
bf917e64bf27e4ec49623704a7c621b32f05e1c7bf1b89960c82d4203c4efa6a1056
e083be789d017f
EvaluationElement = 0300fadd09cc84c7e91c2173be0e65ee3c1b6ef98cf0cdd7
57ec432f12f13b5d457edc0311d61ddd831f8becd5231bdc492e92c9d0c103e55ea5
16ee00fe64c10d0e8e,03005941fb7eabc6353a5c2e4a6b284b7b8ee8da6c4435af6
c4c472195bb0deb44e7dc215299c7fe38feafa2b0a1a1db7dd3090c5ce8171247f64
7da20e04acef8164d
Proof = 0186deebd9e2db71ca43bcb57311371390c2d04ac9c3189e155f10c9c548
f6f22c051d38203493176e8392ef405c783759c735cb6f7219636c140cb2dc070a11
58c001bfdb12f34e00a582e22e1eb2fbdebcffe4b6e0818de5c50451617213e1ab20
fa5392eb3b535206140a2619732c012d5f331a615755f5397feb9e1fb16d1320d20d
ProofRandomScalar = 010a82559ee5e4ba79c390c4033405e3f792bc49daa905c6
94707e7e0191104b34d68c7cc81c2e392da60b838eadf434b693d9b4f7c7beb31e37
008156656c19382b
Output = 70ad5e29de9f6e35f16afab3b97c1b26fdf6be0da60aff48a99980ddb8d
7c2d728a8a5d2837179bfddd612712e014c0c9b9596cbb5a6ee6761c564dbb8921b4
e,ee2d8e42030da6283ab59a11f41a171c65e208306e00c6f965a56c10f33bf0942b
b38b7e1a33c70bc3542d27220379cbcef8b91898c720be948e9db214a14bb9
```

Authors' Addresses

Alex Davidson
Brave Software

Email: alex.davidson92@gmail.com

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

Email: armfazh@cloudflare.com

Nick Sullivan
Cloudflare, Inc.
101 Townsend St

San Francisco,
United States of America

Email: nick@cloudflare.com

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

Email: caw@heapingbits.net