Workgroup: Network Working Group
Internet-Draft: draft-irtf-cfrg-voprf-14
Published: 6 October 2022
Intended Status: Informational
Expires: 9 April 2023
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 is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 9 April 2023.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction
  1.1. Change log
  1.2. Requirements
  1.3. Notation and Terminology
2. Preliminaries
  2.1. Prime-Order Group
  2.2. Discrete Logarithm Equivalence Proofs
    2.2.1. Proof Generation
    2.2.2. Proof Verification
3. Protocol
  3.1. Configuration
  3.2. Key Generation and Context Setup
  3.3. Online Protocol
    3.3.1. OPRF Protocol
    3.3.2. VOPRF Protocol
    3.3.3. POPRF Protocol
4. Ciphersuites
  4.1. Ciphersuite Registry
    4.1.1. OPRF(ristretto255, SHA-512)
    4.1.2. OPRF(decaf448, SHAKE-256)
    4.1.3. OPRF(P-256, SHA-256)
    4.1.4. OPRF(P-384, SHA-384)
    4.1.5. OPRF(P-521, SHA-512)
  4.2. Input Validation
    4.2.1. Element Validation
    4.2.2. Scalar Validation
  4.3. Future Ciphersuites
5. Application Considerations
  5.1. Input Limits
  5.2. External Interface Recommendations

5.3. Error Considerations
5.4. POPRF Public Input
6. Security Considerations
6.1. Security Properties
6.2. Security Assumptions
6.2.1. OPRF and VOPRF Assumptions
6.2.2. POPRF Assumptions
6.2.3. Static Diffie Hellman Attack and Security Limits
6.3. Domain Separation
6.4. Timing Leaks
7. Acknowledgements
8. References
8.1. Normative References
8.2. Informative References
Appendix A. Test Vectors
A.1. OPRF(ristretto255, SHA-512)
A.1.1. OPRF Mode
A.1.2. VOPRF Mode
A.1.3. POPRF Mode
A.2. OPRF(decaf448, SHAKE-256)
A.2.1. OPRF Mode
A.2.2. VOPRF Mode
A.2.3. POPRF Mode
A.3. OPRF(P-256, SHA-256)
A.3.1. OPRF Mode
A.3.2. VOPRF Mode
A.3.3. POPRF Mode
A.4. OPRF(P-384, SHA-384)
A.4.1. OPRF Mode
A.4.2. VOPRF Mode
A.4.3. POPRF Mode
A.5. OPRF(P-521, SHA-512)
A.5.1. OPRF Mode
A.5.2. VOPRF Mode
A.5.3. POPRF Mode
Authors' Addresses

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 OPRFs 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-14:

- Correct current state of formal analysis for the VOPRF protocol variant.

draft-13:

- Editorial improvements based on Crypto Panel Review.

draft-12:

- Small editorial fixes

draft-11:

- Change Evaluate to BlindEvaluate, and add Evaluate for PRF evaluation

draft-10:

- Editorial improvements

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(x, xLen): Converts a non-negative integer x into a byte array of specified length xLen as described in [RFC8017]. Note that this function returns a byte array in big-endian byte order.

*The notation T U[N] refers to an array called U containing N items of type T. The type opaque means one single byte of

uninterpreted data. Items of the array are zero-indexed and referred as U[j] such that 0 <= j < N.

All algorithms and procedures described in this document are laid out in a Python-like pseudocode. Each function takes a set of inputs and parameters and produces a set of output values. Parameters become constant values once the protocol variant and the ciphersuite are fixed.

The PrivateInput data type refers to inputs that are known only to the client in the protocol, whereas the PublicInput data type refers to inputs that are known to both client and server in the protocol. Both PrivateInput and PublicInput are opaque byte strings of arbitrary length no larger than 2^13 octets.

String values such as "DeriveKeyPair", "Seed-", and "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. See Section 4 for specific instances of groups.
- *Hash: A cryptographic hash function whose output length is Nh bytes.

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. In

prime-order groups, any element can generate the other elements of the group. Usually, one element is fixed and defined as the group generator. 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.)

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. The case when the scalar multiplication is performed on the group generator is denoted as ScalarMultGen(r). Given two elements A and B, the discrete logarithm problem is to find an integer k such that B = k*A. Thus, k is the discrete logarithm of B with respect to the base A. The set of scalars corresponds to GF(p), a prime field of order p, and are represented as the set of integers defined by {0, 1, ..., p-1}. 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).

- *Generator(): Outputs the generator element of the group.

- *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): Returns the inverse of input Scalar s on GF(p).

*SerializeElement(A): A member function of Group that maps a group element A to a unique byte array buf of fixed length Ne bytes.

*DeserializeElement(buf): A member function of Group that maps a byte array buf to a group element A, or raise a DeserializeError if the input is not a valid byte representation of an element. See Section 4.2 for further requirements on input validation.

*SerializeScalar(s): A member function of Group that maps a scalar element s to a unique byte array buf of fixed length Ns bytes.

*DeserializeScalar(buf): A member function of Group that maps a byte array buf to a scalar s, or raise a DeserializeError if the input is not a valid byte representation of a scalar. See Section 4.2 for further requirements on input validation.

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 Logarithm Equivalence Proofs

A proof of knowledge allows a prover to convince a verifier that some statement is true. If the prover can generate a proof without interaction with the verifier, the proof is noninteractive. If the verifier learns nothing other than whether the statement claimed by the prover is true or false, the proof is zero-knowledge.

This section describes a noninteractive zero-knowledge proof for discrete logarithm equivalence (DLEQ). A DLEQ proof demonstrates that two pairs of group elements have the same discrete logarithm without revealing the discrete logarithm.

The DLEQ proof resembles the Chaum-Pedersen [ChaumPedersen] proof, which is shown to be zero-knowledge by Jarecki, et al. [JKK14] and is noninteractive after applying the Fiat-Shamir transform [FS00]. Furthermore, Davidson, et al. [DGSTV18] showed a proof system for batching DLEQ proofs that has constant-size proofs with respect to the number of inputs. The specific DLEQ proof system presented below follows this latter construction with two modifications: (1) the

transcript used to generate the seed includes more context information, and (2) the individual challenges for each element in the proof is derived from a seed-prefixed hash-to-scalar invocation rather than being sampled from a seeded PRNG. The description 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. Given elements A and B, two non-empty lists of elements C and D of length m, and a scalar k; this function produces a proof that k*A == B and k*C[i] == D[i] for each i in [0, ..., m - 1]. The output is a value of type Proof, which is a tuple of two Scalar values.

GenerateProof accepts lists of inputs to amortize the cost of proof generation. Applications can take advantage of this functionality to produce a single, constant-sized proof for m DLEQ inputs, rather than m proofs for m DLEQ inputs.

```
Input:

  Scalar k
  Element A
  Element B
  Element C[m]
  Element D[m]

Output:

  Proof proof

Parameters:

  Group G

def GenerateProof(k, A, B, C, D)
  (M, Z) = ComputeCompositesFast(k, B, C, D)

  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

  return [c, s]
```

The helper function ComputeCompositesFast is as defined below, and is an optimization of the ComputeComposites function for servers since they have knowledge of the private key.

```
Input:

  Scalar k
  Element B
  Element C[m]
  Element D[m]

Output:

  Element M
  Element Z

Parameters:

  Group G
  PublicInput contextString

def ComputeCompositesFast(k, B, C, D):
  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 in range(m):
    Ci = G.SerializeElement(C[i])
    Di = G.SerializeElement(D[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 * C[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 3.2.

2.2.2. Proof Verification

Verifying a proof is done with the VerifyProof function, defined below. This function takes elements A and B, two non-empty lists of elements C and D of length m, and 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. Note this

function can verify proofs on lists of inputs whenever the proof was generated as a batched DLEQ proof with the same inputs.

```
Input:

  Element A
  Element B
  Element C[m]
  Element D[m]
  Proof proof

Output:

  boolean verified

Parameters:

  Group G

def VerifyProof(A, B, C, D, proof):
  (M, Z) = ComputeComposites(B, C, D)
  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)
  verified = (expectedC == c)

  return verified
```

The definition of ComputeComposites is given below.

```
Input:

  Element B
  Element C[m]
  Element D[m]

Output:

  Element M
  Element Z

Parameters:

  Group G
  PublicInput contextString

def ComputeComposites(B, C, D):
  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 in range(m):
    Ci = G.SerializeElement(C[i])
    Di = G.SerializeElement(D[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 * C[i] + M
    Z = di * D[i] + Z

  return (M, Z)
```

When used in the protocol described in Section 3, the parameter contextString is as defined in Section 3.2.

3. Protocol

In this section, we define three protocol variants referred as the OPRF, VOPRF, and POPRF modes with the following properties.

In the OPRF 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. After the

execution of the protocol, the client learns output and the server learns nothing. This interaction is shown below.

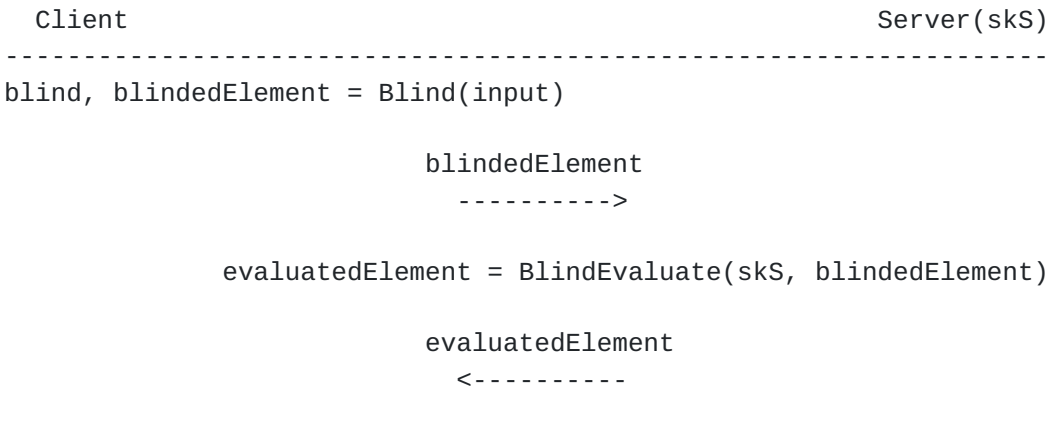


Figure 1: OPRF protocol overview

In the VOPRF mode, the client additionally receives proof that the server used skS in computing the function. To achieve verifiability, as in [JKK14], the server provides a zero-knowledge proof that the key provided as input by the server in the BlindEvaluate 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.

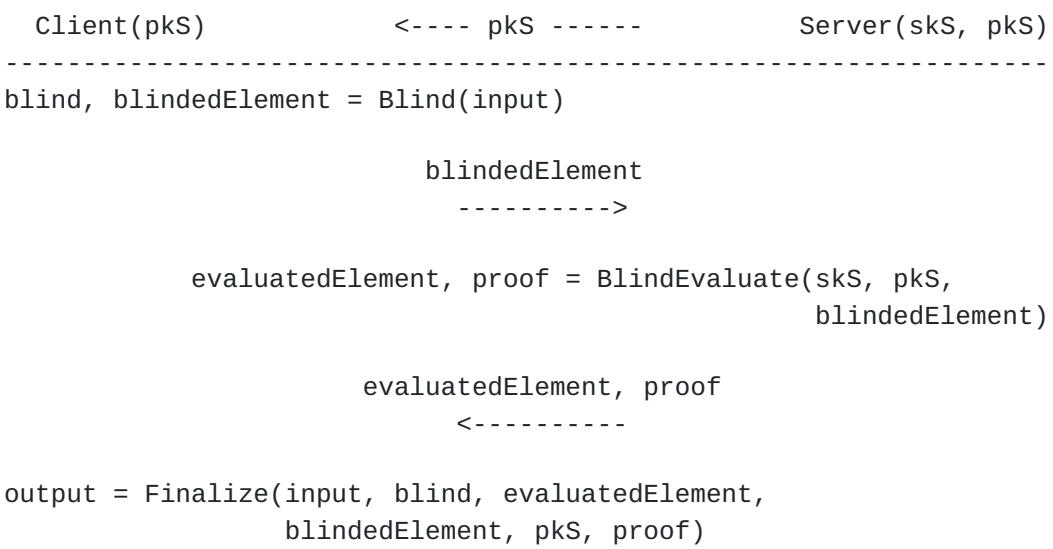


Figure 2: VOPRF protocol overview with additional proof

The POPRF 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) as is shown below.

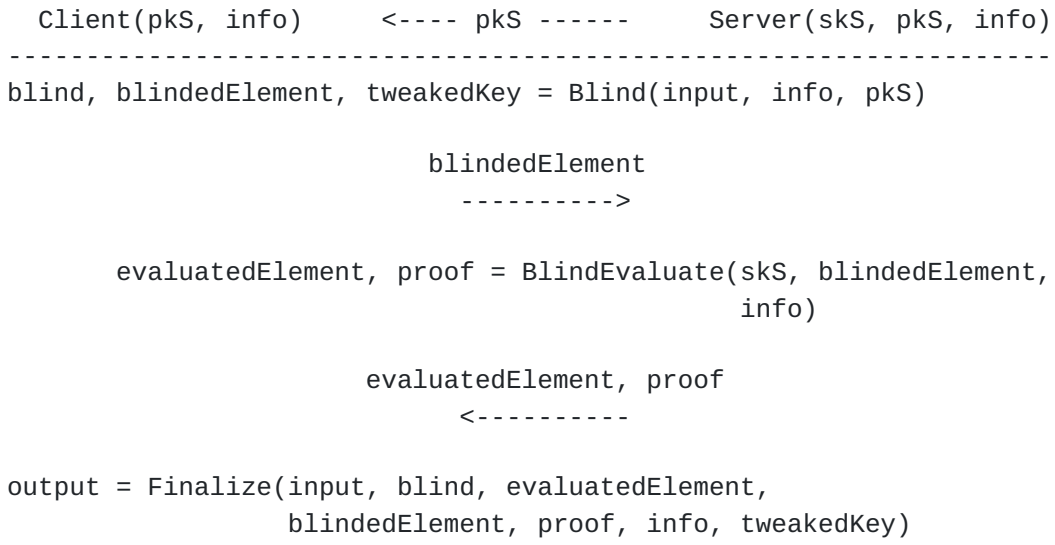


Figure 3: POPRF protocol overview with additional public input

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 (in hexadecimal):

Mode	Value
modeOPRF	0x00
modeVOPRF	0x01
modePOPRF	0x02

Table 1: Identifiers for OPRF modes

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 "VOPRF10-" || I2OSP(mode, 1) || I2OSP(suiteID, 2)
```

[[RFC editor: please change "VOPRF10" 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, the server key pair (skS, pkS) is generated using the following function, which accepts a randomly generated seed of length Ns bytes and an optional (and possible empty) public info string. The constant Ns corresponds to the size in bytes 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):
  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.ScalarMultGen(skS)
  return skS, pkS
```

Also during 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 context, such as OPRFServerContext, is an implementation-specific data structure that stores a context string and the relevant key material for each party.

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. Elements and scalars are serialized to byte arrays, and values of type Proof are serialized as the concatenation of two serialized scalars. 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 after deserializing Element values; 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()
  inputElement = G.HashToGroup(input)
  if inputElement == G.Identity():
    raise InvalidInputError
  blindedElement = blind * inputElement

  return blind, blindedElement
```

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

```
Input:

  Scalar skS
  Element blindedElement

Output:

  Element evaluatedElement

def BlindEvaluate(skS, blindedElement):
  evaluatedElement = skS * blindedElement
  return evaluatedElement
```

Servers send the output evaluatedElement to clients for processing. Recall that servers may process multiple client inputs by applying the BlindEvaluate function to each blindedElement received, and returning an array with the corresponding evaluatedElement values.

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)
```

Servers can compute the PRF result using a given input using the following Evaluate function.

```
Input:

  Scalar skS
  PrivateInput input

Output:

  opaque output[Nh]

Parameters:

  Group G

Errors: InvalidInputError

def Evaluate(skS, input):
  inputElement = G.HashToGroup(input)
  if inputElement == G.Identity():
    raise InvalidInputError
  evaluatedElement = skS * inputElement
  issuedElement = G.SerializeElement(evaluatedElement)

  hashInput = I2OSP(len(input), 2) || input ||
              I2OSP(len(issuedElement), 2) || issuedElement ||
              "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 BlindEvaluate function.

```
Input:

  Scalar skS
  Element pkS
  Element blindedElement

Output:

  Element evaluatedElement
  Proof proof

Parameters:

  Group G

def BlindEvaluate(skS, pkS, blindedElement):
  evaluatedElement = skS * blindedElement
  blindedElements = [blindedElement]     // list of length 1
  evaluatedElements = [evaluatedElement] // list of length 1
  proof = GenerateProof(skS, G.Generator(), pkS,
                        blindedElements, evaluatedElements)
  return evaluatedElement, proof
```

In the description above, inputs to GenerateProof are one-item lists. Using larger lists allows servers to batch the evaluation of multiple elements while producing a single batched DLEQ proof for them.

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
  Element pkS
  Proof proof

Output:

  opaque output[Nh]

Parameters:

  Group G

Errors: VerifyError

def Finalize(input, blind, evaluatedElement,
             blindedElement, pkS, proof):
  blindedElements = [blindedElement]     // list of length 1
  evaluatedElements = [evaluatedElement] // list of length 1
  if VerifyProof(G.Generator(), pkS, blindedElements,
                 evaluatedElements, 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)
```

As in BlindEvaluate, inputs to VerifyProof are one-item lists. Clients can verify multiple inputs at once whenever the server produced a batched DLEQ proof for them.

Finally, servers can compute the PRF result using a given input using the Evaluate function described in Section 3.3.1.

3.3.3. POPRF Protocol

The POPRF protocol begins with the client blinding its input, using the following modified Blind function. In this step, the client also binds a public info value, which produces an additional tweakedKey to be used later in the protocol. 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, if not detected at this point, will cause server evaluation to fail. Dealing with either failure is an application-specific decision; see Section 5.3.

```
Input:

  PrivateInput input
  PublicInput info
  Element pkS

Output:

  Scalar blind
  Element blindedElement
  Element tweakedKey

Parameters:

  Group G

Errors: InvalidInputError

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

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

  blindedElement = blind * inputElement

  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 BlindEvaluate function.

```
Input:

  Scalar skS
  Element blindedElement
  PublicInput info

Output:

  Element evaluatedElement
  Proof proof

Parameters:

  Group G

Errors: InverseError

def BlindEvaluate(skS, 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.ScalarMultGen(t)
  evaluatedElements = [evaluatedElement] // list of length 1
  blindedElements = [blindedElement]     // list of length 1
  proof = GenerateProof(t, G.Generator(), tweakedKey,
                        evaluatedElements, blindedElements)

  return evaluatedElement, proof
```

In the description above, inputs to GenerateProof are one-item lists. Using larger lists allows servers to batch the evaluation of multiple elements while producing a single batched DLEQ proof for them.

BlindEvaluate triggers InverseError when the function is about to calculate the inverse of a zero scalar, which does not exist and therefore yields a failure in the protocol. This only occurs for info values that map to the secret key of the server. Thus, clients that observe this signal are assumed to know the server secret key. Hence, this error can be a signal for the server to replace its secret key.

The server sends both evaluatedElement and proof back to the client. Upon receipt, the client processes both values to complete the POPRF 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

Errors: VerifyError

def Finalize(input, blind, evaluatedElement, blindedElement,
             proof, info, tweakedKey):
  evaluatedElements = [evaluatedElement] // list of length 1
  blindedElements = [blindedElement]     // list of length 1
  if VerifyProof(G.Generator(), tweakedKey, evaluatedElements,
                 blindedElements, 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)
```

As in BlindEvaluate, inputs to VerifyProof are one-item lists. Clients can verify multiple inputs at once whenever the server produced a batched DLEQ proof for them.

Finally, servers can compute the PRF result using a given input using the Evaluate function described below.

```
Input:

  Scalar skS
  PrivateInput input
  PublicInput info

Output:

  opaque output[Nh]

Parameters:

  Group G

Errors: InvalidInputError, InverseError

def Evaluate(skS, input, info):
  inputElement = G.HashToGroup(input)
  if inputElement == G.Identity():
    raise InvalidInputError

  framedInfo = "Info" || I2OSP(len(info), 2) || info
  m = G.HashToScalar(framedInfo)
  t = skS + m
  if t == 0:
    raise InverseError
  evaluatedElement = G.ScalarInverse(t) * inputElement
  issuedElement = G.SerializeElement(evaluatedElement)

  hashInput = I2OSP(len(input), 2) || input ||
              I2OSP(len(info), 2) || info ||
              I2OSP(len(issuedElement), 2) || issuedElement ||
              "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 the generator element 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 Group.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 Group.Order() and in little-endian order.

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

*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 Group.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 Group.Order() and in little-endian order.

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

*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 Group.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 Group.Order() and in big-endian order.

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

*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 Group.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 Group.Order() and in big-endian order.

*Hash: SHA-384, and Nh = 48 bytes.

*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 Group.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 Group.Order() and in big-endian order.

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

*ID: 0x0005

4.2. Input Validation

Since messages are serialized before transmission between client and server, deserialization is followed by input validation to prevent malformed or invalid inputs from being used in the protocol. 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 input validation of elements and scalars is implemented for all prime-order groups included in the above ciphersuite list.

4.2.1. Element Validation

Recovering a group element from an arbitrary byte array must validate 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, it is required to perform 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 identity. If these checks fail, validation returns an InputValidationError.

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 or if the decoded element is the identity, validation returns an InputValidationError.

4.2.2. Scalar 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 Group.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 Group.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

In Section 3.3, the interface of the protocol functions allows that some inputs (and outputs) to be group elements and scalars. However, implementations can instead operate over group elements and scalars internally, and only expose interfaces that operate with an application-specific format of messages.

5.3. Error Considerations

Some OPRF variants specified in this document have fallible operations. For example, Finalize and BlindEvaluate can fail if any element received from the peer fails input validation. 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.

* InputValidationError: Validation of byte array inputs failed; Section 4.2.

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

VOPRF 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: For a random sampling of k, 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.

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 computing power cannot recover any information about the client's private input x from an invocation of the protocol.

Essentially, input secrecy is the property that, even if the server learns the client's private input x at some point in the future, the server cannot link any particular PRF evaluation to x. This property is also known as unlinkability [DGSTV18].

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 output it computes is correct. This is taken with respect to the private 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 client and server must be able to perform the PRF on client's private input and public input. 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.

This property becomes useful when dealing with key management operations such as the rotation of server's keys.

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 particular, the VOPRF construction is similar to the [JKK14] construction with the following distinguishing properties:

1. This document does not use session identifiers to differentiate different instances of the protocol; and

2. This document supports batching so that multiple evaluations can happen at once whilst only constructing one DLEQ proof object. This is enabled using an established batching technique [DGSTV18].

The pseudorandomness and input secrecy (and verifiability) of the OPRF (and VOPRF) protocols in [JKK14] are based on an assumption with oracle access to the Computational Diffie Hellman (CDH) assumption, known as the One-More Gap CDH, that is computationally difficult to solve in the corresponding prime-order group. [JKK14] proves these properties for one instance (i.e., one key) of the VOPRF protocol, and without batching. There is currently no security analysis available for the VOPRF protocol described in this document in a setting with multiple server keys or batching.

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 DLEQ proof object. This is enabled using an established batching technique [DGSTV18].

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. Tyagi et al. [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 BlindEvaluate 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 BlindEvaluate 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 BlindEvaluate 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 BlindEvaluate.

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-Hernández, A., Valdez, S., and C. A. Wood, "Privacy Pass Issuance Protocol", Work in Progress, Internet-Draft, draft-ietf-privacypass-protocol-06, 6 July 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-privacypass-protocol-06>.

[I-D.irtf-cfrg-hash-to-curve] Faz-Hernández, 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-16, 15 June 2022, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hash-to-curve-16>.

[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-09, 6 July 2022, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-opaque-09>.

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

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

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

[RISTRETTO] de Valence, H., Grigg, J., Hamburg, M., Lovecruft, I., Tankersley, G., and F. Valsorda, "The ristretto255 and decaf448 Groups", Work in Progress, Internet-Draft, draft-irtf-cfrg-ristretto255-decaf448-03, 25 February 2022, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-ristretto255-decaf448-03>.

8.2. Informative References

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

[ChaumPedersen] Chaum, D. and T. Pedersen, "Wallet Databases with Observers", Advances in Cryptology - CRYPTO' 92 pp. 89-105, DOI 10.1007/3-540-48071-4_7, August 2007, <https://doi.org/10.1007/3-540-48071-4_7>.

[Cheon06] Cheon, J., "Security Analysis of the Strong Diffie-Hellman Problem", Advances in Cryptology - EUROCRYPT 2006 pp. 1-11, DOI 10.1007/11761679_1, 2006, <https://doi.org/10.1007/11761679_1>.

[DGSTV18] Davidson, A., Goldberg, I., Sullivan, N., Tankersley, G., and F. Valsorda, "Privacy Pass: Bypassing Internet Challenges Anonymously", Proceedings on Privacy Enhancing Technologies vol. 2018, no. 3, pp. 164-180, DOI 10.1515/popets-2018-0026, April 2018, <https://doi.org/10.1515/popets-2018-0026>.

[FS00] Fiat, A. and A. Shamir, "How To Prove Yourself: Practical Solutions to Identification and Signature Problems", Advances in Cryptology - CRYPTO' 86 pp. 186-194, DOI 10.1007/3-540-47721-7_12, April 2007, <https://doi.org/10.1007/3-540-47721-7_12>.

[JKK14] Jarecki, S., Kiayias, A., and H. Krawczyk, "Round-Optimal Password-Protected Secret Sharing and T-PAKE in the Password-Only Model", Lecture Notes in Computer Science pp. 233-253, DOI 10.1007/978-3-662-45608-8_13, 2014, <https://doi.org/10.1007/978-3-662-45608-8_13>.

[JKKX16] Jarecki, S., Kiayias, A., Krawczyk, H., and J. Xu, "Highly-Efficient and Composable Password-Protected Secret Sharing (Or: How to Protect Your Bitcoin Wallet Online)", 2016 IEEE European Symposium on Security and Privacy (EuroS&P), DOI 10.1109/eurosp.2016.30, March 2016, <https://doi.org/10.1109/eurosp.2016.30>.

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

[PrivacyPass] "Privacy Pass", <https://github.com/privacypass/team>.

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

[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] Shirvanian, M., Jarecki, S., Krawczyk, H., and N. Saxena, "SPHINX: A Password Store that Perfectly Hides Passwords from Itself", In 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), DOI 10.1109/ICDCS.2017.64, June 2017, <https://doi.org/10.1109/ICDCS.2017.64>.

[TCRSTW21] Tyagi, N., Celi, S., Ristenpart, T., Sullivan, N., Tessaro, S., and C. Wood, "A Fast and Simple Partially Oblivious PRF, with Applications", Advances in Cryptology - EUROCRYPT 2022 pp. 674-705, DOI 10.1007/978-3-031-07085-3_23, 2022, <https://doi.org/10.1007/978-3-031-07085-3_23>.

[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 fields of each test vector are described below.

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

- "Info": The public info, an opaque byte string. Only present for POPRF test 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 BlindEvaluate(), a serialized Element of Ne bytes long.

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

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

- "Output": The protocol output, an opaque 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 = e617ae6f2d10de61e16cab73023c5a2df74335d13f89470957214664468d2
e0b
```

A.1.1.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = 64d37aed22a27f5191de1c1d69fadb899d8862b58eb4220029e036ec4c1f
6706
BlindedElement = c83d0d8a3e80be2ced8bf35c5f3e24d42260ca8fa9a0403ca83
033588c26614d
EvaluationElement = b29ca44d6dfafc77a50b72abc53cfb7abcbe9cf6714afc76
893ee8dcaf053b59
Output = 8a19c9b8f4459d541ebbfff4e29f36620e44e825a27b0f2e3a3c0d8e963
588ee04348312dc8b43a48c41d4e7d904f95c91813a6b4f624392433f0568409da62
8
```

A.1.1.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 64d37aed22a27f5191de1c1d69fadb899d8862b58eb4220029e036ec4c1f
6706
BlindedElement = 8673ffd2f26b2579922fc485c77e106def00982e0abb233b4c6
e54841d43ba29
EvaluationElement = 68ed7037846f48a1b4073a0d110f6e4de8f53ab845365c0f
3d7f1b67caa39126
Output = bcdbd421c0863495d63d81a868858f34f5215437c5777072a92703f36b3
6c4a2d3e7e54a5762e70b06223527c211e2d4364481270f72971a2db8b7ab8fad84e
e
```

A.1.2. VOPRF Mode

```
Seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3
3a3
KeyInfo = 74657374206b6579
skSm = a3b8dea4a99be2469da7f7d2d93fe5f2867317d6705350475d47739c7214d
a07
pkSm = c00fbee6832a8e5d6cc1d1a23315daf6a6018f19e29ba37b05499259da854
b48
```

A.1.2.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = 64d37aed22a27f5191de1c1d69fadb899d8862b58eb4220029e036ec4c1f
6706
BlindedElement = 6cce2c7913f4c8c0ac44ec149a1544b0e711e1630753d4efc7c
5fe36a4d50638
EvaluationElement = 826f2f3e553a039bcd69c9df6cb166e7943fd207089ae704
1f6041322ce7033a
Proof = 2e541a6962e783d2f42d5f4fb1364e51c368e95e83a962614714e9dfe21a
720cd8c8eb8106131b4a758b5a0987d3870adb348f5eae7b4a2bc26735928cc4b90c
ProofRandomScalar = 222a5e897cf59db8145db8d16e597e8facb80ae7d4e26d98
81aa6f61d645fc0e
Output = 4d5dd83db5bfd850e3e0c17519f1013aab904e7b131dc1ded31f7a76aac
f040f6b344b0e635cf6df30771a35157e0e3d9539f7a891b48cd8521692b15c51538
d
```

A.1.2.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 64d37aed22a27f5191de1c1d69fadb899d8862b58eb4220029e036ec4c1f
6706
BlindedElement = 6a4e632b76a2cfcb0295ee74098a15a3e858f6006fd9fa8576a
5813e051ac134
EvaluationElement = 2cb879d933a1af46c77e89f3f39a38f80347bf4716da3dc3
07c8aa1282179823
Proof = eabae3489c46b9e9a8da0cc921d2bc2960ef5fb0b38c8f067cc5c21f62f4
eb0ff5472009aec126f543b6051b5d62ccbf2625aab6684076c26cfdf0904257090c
ProofRandomScalar = 222a5e897cf59db8145db8d16e597e8facb80ae7d4e26d98
81aa6f61d645fc0e
Output = 5c3fe06ef39905710a124df0727c6c938f48234b35ccc4548c0736d7f6f
36e6b7333a9aefc93d6b1ee20151a40bce453866b62cf5d41799982fee6100680915
9
```

A.1.2.3. Test Vector 3, Batch Size 2

```
Input = 00,5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 64d37aed22a27f5191de1c1d69fadb899d8862b58eb4220029e036ec4c1f
6706,222a5e897cf59db8145db8d16e597e8facb80ae7d4e26d9881aa6f61d645fc0
e
BlindedElement = 6cce2c7913f4c8c0ac44ec149a1544b0e711e1630753d4efc7c
5fe36a4d50638,aa9908e4c40b7fe5f091cf0f7fb8ec75ffdaaf2d19512b7b9939f0
ffaaa0654f
EvaluationElement = 826f2f3e553a039bcd69c9df6cb166e7943fd207089ae704
1f6041322ce7033a,902ef95488cc3c47fe569bc96c922a4ae3f9ebd8ccbc71bfefa
5f1e7da9ab953
Proof = d9bfee92cd7496cdf469947b534549ceb79ebd7b5695d20437b3e14758cf
de0998eaa13a480cc35b562cbfb1412b1677650cd901b5fb4d6805581a95b440320f
ProofRandomScalar = 419c4f4f5052c53c45f3da494d2b67b220d02118e0857cdb
cf037f9ea84bbe0c
Output = 4d5dd83db5bfd850e3e0c17519f1013aab904e7b131dc1ded31f7a76aac
f040f6b344b0e635cf6df30771a35157e0e3d9539f7a891b48cd8521692b15c51538
d,5c3fe06ef39905710a124df0727c6c938f48234b35ccc4548c0736d7f6f36e6b73
33a9aefc93d6b1ee20151a40bce453866b62cf5d41799982fee61006809159
```

A.1.3. POPRF Mode

```
Seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3
KeyInfo = 74657374206b6579
skSm = 024eaeb72e5b3729d7f19d90aa44e3d2f4c445fb29011ffd755655636f2b1
00a
pkSm = e001954ccd18ec5aa89bcbf26c03d84dc4d9c9b973d9f06b1e0ceb7b79f41
d65
```

A.1.3.1. Test Vector 1, Batch Size 1

```
Input = 00
Info = 7465737420696e666f
Blind = 64d37aed22a27f5191de1c1d69fadb899d8862b58eb4220029e036ec4c1f
6706
BlindedElement = 009ffa1ffc529e4f1d3d8de1c06d22fbb15e39920a72ad4efed
6c39af9438a2d
EvaluationElement = aa9af25bf4edead5e2e0a4b8f93db9b497017f93cf68c750
45f02172bfc5d304
Proof = bb893ccce54685a871185bb056cb5e0594d09d3b53f2f879de06a650b8ae
ff08371f2ff9f3d5cac7f393cc37b2c71c2a6fbb80f35fe36b8e5cbddf11469c8e03
ProofRandomScalar = 222a5e897cf59db8145db8d16e597e8facb80ae7d4e26d98
81aa6f61d645fc0e
Output = e7ed59e3f808c369598961ebfd9af74272894e0904d1c11653a21b08204
dba1a5fb5c3dd6be6c419190a84b576d91eb3d8d920d450fee0427fd24524950d72d
6
```

A.1.3.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Info = 7465737420696e666f
Blind = 64d37aed22a27f5191de1c1d69fadb899d8862b58eb4220029e036ec4c1f
6706
BlindedElement = 5e009e08e228f95ee3703cff60a1d54225bb282bdb6d7dc9a78
e287f8418315a
EvaluationElement = 2e528236481eb6d87b07ef5f8c17910323d04b3bf0cb2f2d
23d5a7ad9f069b22
Proof = 3796381ab287189839288bbaffc971eb87c3a28226fa99dc83b363adb2f4
b20e4ae81fb675ebcd43d13918f71846cb488d0ce7d473bfca68450a5a5472564500
ProofRandomScalar = 222a5e897cf59db8145db8d16e597e8facb80ae7d4e26d98
81aa6f61d645fc0e
Output = 9a0d8c55e2fef4bada9fb5877a0e739496e539a0d835722911dab9ec112
397e763a605acbc072619e8b8acefb8ee704a357556edc802648089d684baa763ce1
4
```

A.1.3.3. Test Vector 3, Batch Size 2

```
Input = 00,5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Info = 7465737420696e666f
Blind = 64d37aed22a27f5191de1c1d69fadb899d8862b58eb4220029e036ec4c1f
6706,222a5e897cf59db8145db8d16e597e8facb80ae7d4e26d9881aa6f61d645fc0
e
BlindedElement = 009ffa1ffc529e4f1d3d8de1c06d22fbb15e39920a72ad4efed
6c39af9438a2d,1ee64b9e5148987ca6647ccddc11ef506231e986d5ce08ef9b8230
871f840b3a
EvaluationElement = aa9af25bf4edead5e2e0a4b8f93db9b497017f93cf68c750
45f02172bfc5d304,3073794fd68f64432b4d1f24752c4398f0e81e00b5b5842e463
5dd381331091b
Proof = 7d59db67715a9030d46ab50a614fb55927961c8d9322cb6973ef36775309
810b9f4a670ba4b9321f5cf753be2a58dee0730cfabd12b8f25a8a342e158ae2b608
ProofRandomScalar = 419c4f4f5052c53c45f3da494d2b67b220d02118e0857cdb
cf037f9ea84bbe0c
Output = e7ed59e3f808c369598961ebfd9af74272894e0904d1c11653a21b08204
dba1a5fb5c3dd6be6c419190a84b576d91eb3d8d920d450fee0427fd24524950d72d
6,9a0d8c55e2fef4bada9fb5877a0e739496e539a0d835722911dab9ec112397e763
a605acbc072619e8b8acefb8ee704a357556edc802648089d684baa763ce14
```

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

A.2.1. OPRF Mode

```
Seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3
KeyInfo = 74657374206b6579
skSm = 30f71e5b5be9c91dd54c5a48e82be8d47eeb2cb2c45d7874a45dddc85af8d
3f95b1ce73a99c47edc26ac9ddd936bd9b6b73728995bf1d213
```

A.2.1.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = 64d37aed22a27f5191de1c1d69fadb899d8862b58eb4220029e036ec65fa
3833a26e9388336361686ff1f83df55046504dfecad8549ba112
BlindedElement = a4205d2af0410dccbd4464629ba1b835456d04d994cf93988cf
2c3b9d45d3c4671c7625f52c66c760a069e2c3c367826debb13da089d735c
EvaluationElement = e8d78cf5212fddf940f9f6fe02250ed83cc0595e3f0e7481
1cdb9f62c0fa7fea94c45795637dc5c3ac31ee1cff18d0d675396ae09b302f76
Output = 1c1a9df7d0616e0f5fdfb6479acec73a4f5562da8f9488f3b6112ef11c6
7c5900e0abc3a169486ac7230a306c8796562a045c66305ed7cb2a3fae658e45eae4
c
```

A.2.1.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 64d37aed22a27f5191de1c1d69fadb899d8862b58eb4220029e036ec65fa
3833a26e9388336361686ff1f83df55046504dfecad8549ba112
BlindedElement = ec5b609e5d3c0bb024c35256194694ea6e42aa24d13cf6b0597
49cb36911ccba0923cb73136acdf4bcecf23b6025f7b9b93d2eb0c09d964d
EvaluationElement = 524c3a644e381b4ae416724247f94b996f655167e0d4e1ba
d93cbc731c3beb36e3822e9dcbdc3600966226387a2306ba70eb68db5a64f92f
Output = 95f519e8ff2b54d8d596da2c54829ae3dd900f5c18eef48efa03ef6694c
505bea17b7982246c862d081b9fdcf295debc60abec8b0ddbfdf48bd302a3fe61b21
e
```

A.2.2. VOPRF Mode

```
Seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3
KeyInfo = 74657374206b6579
skSm = 44c46e78aa6386cee57a46c75d124b13ced3e5f055caa3baaad61501330a4
24463400453c97245a8f7b4c65f2c4c3dabd09a049c034f9e20
pkSm = 78f4233110896fd41531fce182094c3bc4cf65f97b23078476b3b68118736
617172d3735c5832081864e7c75cd3ddb449e93068b34ba863e
```

A.2.2.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = 64d37aed22a27f5191de1c1d69fadb899d8862b58eb4220029e036ec65fa
3833a26e9388336361686ff1f83df55046504dfecad8549ba112
BlindedElement = 38b758b69dfaaff8576eaaabfe70801813d95eb098f85516bcd
46a0f68d1ea8cc1dea3bc7c8d340ee77c5bbca6e7d723e51d77e0807acd0d
EvaluationElement = 7a8374bbae55dfc91e10a9d8042015419c505a6a8ac54e5b
93867747eb04252aba316d9f750fa0c54458aa8c90e963a60af5ae6f141af8d2
Proof = 2fd38cf9829c5f3fd294a5eb114356cd67cc5839cf797dc060273e07cf57
0dbabea029f0bf4675d84866865d1d146bfa38eff8195b59cf3c180bab30509061b9
d02e70f709f085dc8c98c0924259c9a3463ef5ceb97105989941155b98bd7b03b1e1
e538850139dc1a56beff1bb9401f
ProofRandomScalar = b1b748135d405ce48c6973401d9455bb8ccd18b01d0295c0
627f67661200dbf9569f73fbb3925daa043a070e5f953d80bb464ea369e5522b
Output = 3db64b6f803391e7c9803135457da250eb29778480c30f29d53e9ff46c3
ce5ba9555418fc28af347c18b77a990eb904d0043a3411837b6d316f749428a9a370
4
```

A.2.2.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 64d37aed22a27f5191de1c1d69fadb899d8862b58eb4220029e036ec65fa
3833a26e9388336361686ff1f83df55046504dfecad8549ba112
BlindedElement = ea9b2d51579f5c07c5c511cf3bba888f5fc76d6ce29075a0b02
5adb3daf4b568045c28e6bd00442251597ba6264e59beaf46220d8405fff6
EvaluationElement = f6d23094a82e33e231003a1ecdd4659029d613932b767451
c607ec428315283fe0b121bf09d7c88cf2ed50910463e38383fb52e5562a87f0
Proof = 104e45c171bd7ca9119af1091e3175c8af4e9efdbd4704b3d5a8dfc99465
9842ea021da27a9c1e0fbac369627eb5e9cf9e82964b7412081f15f6bfc5c68425f6
4f1a4dae420a03d582a6cfffc0fc4da71a145bb5305ae28985e15e067d28523578ea
696205cea28cf5831abed3e40f37
ProofRandomScalar = b1b748135d405ce48c6973401d9455bb8ccd18b01d0295c0
627f67661200dbf9569f73fbb3925daa043a070e5f953d80bb464ea369e5522b
Output = 4dc9ec52b6aa7f1f38a320d10cb58e0d86b040f6376d2f178f42c99986f
e932aca7162cb72dd94056724617979c0f7ea652b1492bbad1d82748a38ff4daf129
8
```

A.2.2.3. Test Vector 3, Batch Size 2

```
Input = 00,5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 64d37aed22a27f5191de1c1d69fadb899d8862b58eb4220029e036ec65fa
3833a26e9388336361686ff1f83df55046504dfecad8549ba112,b1b748135d405ce
48c6973401d9455bb8ccd18b01d0295c0627f67661200dbf9569f73fbb3925daa043
a070e5f953d80bb464ea369e5522b
BlindedElement = 38b758b69dfaaff8576eaaabfe70801813d95eb098f85516bcd
46a0f68d1ea8cc1dea3bc7c8d340ee77c5bbca6e7d723e51d77e0807acd0d,5a788e
f7949021b22da4a4e89b2443458c96fcbec8b66b08df885eec8fb4070fefe8b50e08
5e043c368cc05a9339b5ae31eb6482efc0d933
EvaluationElement = 7a8374bbae55dfc91e10a9d8042015419c505a6a8ac54e5b
93867747eb04252aba316d9f750fa0c54458aa8c90e963a60af5ae6f141af8d2,0ac
81e0e5b9fa6d90be58a6fc3fb4fde57e0efacbe210cebc2c85a6e934114b5e0e5ba4
cc202bde7cd7708415cdcc2312a51fca6ad6f06bf
Proof = a221b134d99ba97cad98bf45341eeacd8a402a6e4c5ea5f93cee54ad0f2b
ee544f67d2859a5253cb9def403bfee9420a5224fad35e3f9a3fbb5f28f6b8abcb34
130beaa158a41d1497aacc2f073b2da5471067bb832ec8044f417f528e2e6ccb897f
992424220d608b5e7bbfd4257e1f
ProofRandomScalar = 63798726803c9451ba405f00ef3acb633ddf0c420574a2ec
6cbf28f840800e355c9fbaac10699686de2724ed22e797a00f3bd93d105a7f23
Output = 3db64b6f803391e7c9803135457da250eb29778480c30f29d53e9ff46c3
ce5ba9555418fc28af347c18b77a990eb904d0043a3411837b6d316f749428a9a370
4,4dc9ec52b6aa7f1f38a320d10cb58e0d86b040f6376d2f178f42c99986fe932aca
7162cb72dd94056724617979c0f7ea652b1492bbad1d82748a38ff4daf1298
```

A.2.3. POPRF Mode

```
Seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3
KeyInfo = 74657374206b6579
skSm = fdd59cb218c7fbdcd48b18ef21ab647a6c210110c765bc3da6c11e563671a
48402c23129ce2ffd021d99da5a2d04158883c65d7f74a4901b
pkSm = 1223e0aec4ee5bc19181078be380cc745d1896e1369aed3cc8a45b40ba3f9
aa1f79e23d542d6529e17465d1954d75e336910c6417de99200
```

A.2.3.1. Test Vector 1, Batch Size 1

```
Input = 00
Info = 7465737420696e666f
Blind = 64d37aed22a27f5191de1c1d69fadb899d8862b58eb4220029e036ec65fa
3833a26e9388336361686ff1f83df55046504dfecad8549ba112
BlindedElement = f86104fcefec6bdca7767bc3e6a2ac9de2b00546579fd50ff66
687df531f7a2dfa8689a6cfdf91efc32d6fff490e722990752b7bc4bda28f
EvaluationElement = 76f27e6fa79cd38638e35f5caa5d641e41526fbfd9272c19
be22dfc8cdd962e6d5d4e0c605c9bd6588eb9698a2bbf792a0827bb1116c8812
Proof = 3a1b3400ad16e1562e731c64520fa5a3664c1487ffe6537e85029842904d
3e01f9e7435b881ab9346847cc3470a2b37e6a10a4ef7bd36b2d06c602086a33252f
39c562aab5820a66c3bdf9d72583587e93ea893725be535cdeca1094d5b4dae119b4
9456162f60034a904f521f7cd818
ProofRandomScalar = b1b748135d405ce48c6973401d9455bb8ccd18b01d0295c0
627f67661200dbf9569f73fbb3925daa043a070e5f953d80bb464ea369e5522b
Output = 2a08f81bf204eb43a57dbc011946861ed715a2fd3d39a3b35e43c74d07d
4734149ba163389a02f6cd33fbb5b84e167d35dca7a7dc00b89418398c255c8293ac
6
```

A.2.3.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Info = 7465737420696e666f
Blind = 64d37aed22a27f5191de1c1d69fadb899d8862b58eb4220029e036ec65fa
3833a26e9388336361686ff1f83df55046504dfecad8549ba112
BlindedElement = e6f508abea28cbb0242f0dae1c0a92e017127edb7c8d8e0ec98
a5ea25c6bc9bb86bfc0bf9b8a086302e29a2a4b0a1d9d80f2d439cfba3ec1
EvaluationElement = 1ea637b039e0ab12c6959c74e275471e33655007a7fa23af
97ec578bcfc8c3381d4929ebf51433b76460d583f16b7cf1e75b9708f5d9d2f7
Proof = d53a1bfeafc5b47fc86406fba080e57434a7004a0739399ccb356f790b13
585da9d69a25c526e039fa06ad6a5781283ea7997eced063fd32e58bc95d57fd771c
ad4a7e23633ae2049eec5ad86ade6a5e98d44f78fd86b5f55ab3c7a03025d6aec1f4
f50a2bd7b9b554841f6b4cd23d14
ProofRandomScalar = b1b748135d405ce48c6973401d9455bb8ccd18b01d0295c0
627f67661200dbf9569f73fbb3925daa043a070e5f953d80bb464ea369e5522b
Output = 80ac73a09fbf8cbd329ff1b7f42d8d14e46ae5b732f776f3203f0680daf
265254360da0afcd9dc1d0cd3858ab21ce8e7a19f0426d7e701cfda34fb8238c9e43
4
```

A.2.3.3. Test Vector 3, Batch Size 2

```
Input = 00,5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Info = 7465737420696e666f
Blind = 64d37aed22a27f5191de1c1d69fadb899d8862b58eb4220029e036ec65fa
3833a26e9388336361686ff1f83df55046504dfecad8549ba112,b1b748135d405ce
48c6973401d9455bb8ccd18b01d0295c0627f67661200dbf9569f73fbb3925daa043
a070e5f953d80bb464ea369e5522b
BlindedElement = f86104fcefec6bdca7767bc3e6a2ac9de2b00546579fd50ff66
687df531f7a2dfa8689a6cfdf91efc32d6fff490e722990752b7bc4bda28f,50c684
9c8f6355687bbc9d4675bcea953cb913c5447c9c8400062ae37f808ce8a75d592c56
f3393d4ea12ec72f9f84402002eb497201089a
EvaluationElement = 76f27e6fa79cd38638e35f5caa5d641e41526fbfd9272c19
be22dfc8cdd962e6d5d4e0c605c9bd6588eb9698a2bbf792a0827bb1116c8812,7ca
a4dd83ecae98fc3e282a0e7df1887393a3fc1e17935dfe355da394756fbfcad65386
eeedf1ba8498411645448c7027753cd9090198c02
Proof = b4f869bf5ec65e0152af5bd29f9fa32c3dfc00355e4e019feda07a281547
fb2f0c559c600bf6cb52a92753264d1c1367e0134b132880732ec70a8c741d60370e
5c22c4aca0e4564732b0157858f3c968bda06aab34c71386ec88afe76ec2c14bf56f
0adf7b05bab826e4aa034cc78837
ProofRandomScalar = 63798726803c9451ba405f00ef3acb633ddf0c420574a2ec
6cbf28f840800e355c9fbaac10699686de2724ed22e797a00f3bd93d105a7f23
Output = 2a08f81bf204eb43a57dbc011946861ed715a2fd3d39a3b35e43c74d07d
4734149ba163389a02f6cd33fbb5b84e167d35dca7a7dc00b89418398c255c8293ac
6,80ac73a09fbf8cbd329ff1b7f42d8d14e46ae5b732f776f3203f0680daf2652543
60da0afcd9dc1d0cd3858ab21ce8e7a19f0426d7e701cfda34fb8238c9e434
```

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

A.3.1. OPRF Mode

```
Seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3
KeyInfo = 74657374206b6579
skSm = 274d7747cf2e26352ecea6bd768c426087da3dfcd466b6841b441ada8412f
b33
```

A.3.1.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = 3338fa65ec36e0290022b48eb562889d89dbfa691d1cde91517fa222ed7a
d364
BlindedElement = 02ff9dc7d4350ab6fe1f41299ec5fa8283b6ef37fc62682ea69
6142e13aad4ae9c
EvaluationElement = 023a5facf92477164f10cc6bf35b4d9272bfadf98dbabbe7
b7a137efa1af6546fb
Output = 488d693c0d43ab75703901fa1398907cf7dc7a90978d1c2f0def63c88e8
1b8b0
```

A.3.1.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 3338fa65ec36e0290022b48eb562889d89dbfa691d1cde91517fa222ed7a
d364
BlindedElement = 03b3cd723330e42975e6e18a6157ecf9455894c18a0189e3e62
4a46d705f790fcc
EvaluationElement = 03f1ea590f2cc4afd45a841285c6be4d88825a9c6c04eb55
a1ca996583dd3e2e9f
Output = dacd8400f6fae62beabead9bc27869b5109fb5d87da338ae2488712ec25
f1be9
```

A.3.2. VOPRF Mode

```
Seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3
3a3
KeyInfo = 74657374206b6579
skSm = b3d12edba73e40401fdc27c0094a56337feb3646d1633345af7e7142a6b15
59d
pkSm = 03f9fc787c9a4dda44a4b811a961d1fd60f87be7465b8a1b9058dc534dae7
0624c
```

A.3.2.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = 3338fa65ec36e0290022b48eb562889d89dbfa691d1cde91517fa222ed7a
d364
BlindedElement = 02bf13d60f3e39e2018c7be9876d88b52e56c0fc2847c8550e3
cee152c51cf72ec
EvaluationElement = 0253e64b5251607348f2b46064805275a849e44db465f649
267c54bd7a774d670f
Proof = d0bff8c87ee38f2b2e9e28161fb0f3bc7e4c3bee7329276487d4fd98d4f4
74fff793a846ffcb44d48f9545e321d89e4e6bccea858089732abf10bf19a220a936
ProofRandomScalar = f9db001266677f62c095021db018cd8cbb55941d4073698c
e45c405d1348b7b1
Output = 9df5d51a9149a86c3660396feabaf790b8c838fc96012adba5acbd913f2
a4016
```

A.3.2.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 3338fa65ec36e0290022b48eb562889d89dbfa691d1cde91517fa222ed7a
d364
BlindedElement = 02a13e263fd9df5aa0078f8d5d6cbe8763e5bee69ee06841a66
dad0db8701480cf
EvaluationElement = 02d9f54fcb97bdab47e6664376a75911f1c3e447f5754550
89d926fbd032cb6e53
Proof = e3ccd78a2f2428d04599c90d4b45e3de49b38a3ba0c80a224b8125747648
718319238dd349cdeb533a6d24333b56aafbb202bec1831511717b231b89b8b36853
ProofRandomScalar = f9db001266677f62c095021db018cd8cbb55941d4073698c
e45c405d1348b7b1
Output = beef8ec835625f610d616d32b1d13f2f899f07c0b8089fa48a1f0ecbc5a
91b8b
```

A.3.2.3. Test Vector 3, Batch Size 2

```
Input = 00,5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 3338fa65ec36e0290022b48eb562889d89dbfa691d1cde91517fa222ed7a
d364,f9db001266677f62c095021db018cd8cbb55941d4073698ce45c405d1348b7b
1
BlindedElement = 02bf13d60f3e39e2018c7be9876d88b52e56c0fc2847c8550e3
cee152c51cf72ec,0322b89e261428d77367cba2aa78fdfa2b21c2919150cafe802e
9020c7f95ec180
EvaluationElement = 0253e64b5251607348f2b46064805275a849e44db465f649
267c54bd7a774d670f,02182b225cfab1d2e25da200549d8b5e2c4581aa7b7bd85be
f9b61a14549f58230
Proof = 900fd64d21320b6059a2810f7046066c4c91a5f4e4f6063c7b51316a4862
2de8f3a28e5f1d0ebe8ae77fdaacbcb1ae92685243e9ceb813bb749dee6c7123270e
ProofRandomScalar = 350e8040f828bf6ceca27405420cdf3d63cb3aef005f40ba
51943c8026877963
Output = 9df5d51a9149a86c3660396feabaf790b8c838fc96012adba5acbd913f2
a4016,beef8ec835625f610d616d32b1d13f2f899f07c0b8089fa48a1f0ecbc5a91b
8b
```

A.3.3. POPRF Mode

```
Seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3
KeyInfo = 74657374206b6579
skSm = 59519f6c7da344f340ad35ad895a5b97437673cc3ac8b964b823cdb52c932
f86
pkSm = 0335065d006a3db4fb09154024dff38c3188a1027e19ce6932e6824c12764
47766
```

A.3.3.1. Test Vector 1, Batch Size 1

```
Input = 00
Info = 7465737420696e666f
Blind = 3338fa65ec36e0290022b48eb562889d89dbfa691d1cde91517fa222ed7a
d364
BlindedElement = 02811b5218bd2bb8361f990efb6062f1201241bcd6f053a5c35
c34dcd7292e7730
EvaluationElement = 02555fc8577c4f88eeb13bc6ac53994f8fb287a33a704592
05ddff91bc19b6a2da
Proof = d87b112dfa11b77f226b85693ab1b5f63adfa491b6e051e570a12392a926
c4816778b527526ba6212c4b0597f13e05f5f9b2223429aab82cd2596625ab1cad0b
ProofRandomScalar = f9db001266677f62c095021db018cd8cbb55941d4073698c
e45c405d1348b7b1
Output = af6525716fe5dd844076bb5cb118ceda08c02c2d1a02368922ddad63f40
f8b44
```

A.3.3.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Info = 7465737420696e666f
Blind = 3338fa65ec36e0290022b48eb562889d89dbfa691d1cde91517fa222ed7a
d364
BlindedElement = 03e9ddbb1fa70461119afcf0ffbfe3fcd105690c14cf0e07872
e72d4f63aa0e197
EvaluationElement = 03156037ca1ab2166e924e6197344a9885256de2cd7d9432
ae36e3f94049e94bbb
Proof = d087b632e2aa4a67e0bc8b7cf012646217a2dfdbf49c60f236a43c66c72b
7f2767b85dc93b96a11e3286ef1ff1864b544a68c2c2d8c2bc35ef7cf7dd34189d3e
ProofRandomScalar = f9db001266677f62c095021db018cd8cbb55941d4073698c
e45c405d1348b7b1
Output = 192f4e5d4f89ffe4b9cea5c1c9619ffe32443a5c04fc35f98c3821420cf
1890c
```

A.3.3.3. Test Vector 3, Batch Size 2

```
Input = 00,5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Info = 7465737420696e666f
Blind = 3338fa65ec36e0290022b48eb562889d89dbfa691d1cde91517fa222ed7a
d364,f9db001266677f62c095021db018cd8cbb55941d4073698ce45c405d1348b7b
1
BlindedElement = 02811b5218bd2bb8361f990efb6062f1201241bcd6f053a5c35
c34dcd7292e7730,0366ff91265bb4a9d24130b9e8cd3ecc523084b512b6b0722de4
4049616b8c374f
EvaluationElement = 02555fc8577c4f88eeb13bc6ac53994f8fb287a33a704592
05ddff91bc19b6a2da,032bdb191ef5604cf43d0c37faead30c4b2b21e3f61c0d47c
cc84850fc5656e500
Proof = 1bd5f64dffa2ab8d6532122887ed55ad17d114020901a7a01cf2412d568e
22b6d0536fd6dbefe9f417060468ee3cc451a8f3750f4d8d4acf1e98437248cc7fa2
ProofRandomScalar = 350e8040f828bf6ceca27405420cdf3d63cb3aef005f40ba
51943c8026877963
Output = af6525716fe5dd844076bb5cb118ceda08c02c2d1a02368922ddad63f40
f8b44,192f4e5d4f89ffe4b9cea5c1c9619ffe32443a5c04fc35f98c3821420cf189
0c
```

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

A.4.1. OPRF Mode

```
Seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3
KeyInfo = 74657374206b6579
skSm = c0503759ddd1e31d8c7eae9304c9b1c16f83d1f6d962e3e7b789cd85fd581
800e96c5c4256131aafcff9a76919abbd55
```

A.4.1.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = 504650f53df8f16f6861633388936ea23338fa65ec36e0290022b48eb562
889d89dbfa691d1cde91517fa222ed7ad364
BlindedElement = 0396a1584fedc4d91ddb753a0c49e0aa2298c1936dbc935d60f
e793d82809f44ff05fbd1922a2cae789d700b5ef4310fb3
EvaluationElement = 0361804cebcb1873cee5e51efd5257cd8b095521cc0089cf
4c1100b1d749e212a044eae6d4f3d852e379eeb1bb54047823
Output = b7ccad41ed7f56be97621bbba8cc3a4f5e8a46a28d72b0fe089d12802f8
6f080b20726e01a99390aba3437ac50c640d6
```

A.4.1.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 504650f53df8f16f6861633388936ea23338fa65ec36e0290022b48eb562
889d89dbfa691d1cde91517fa222ed7ad364
BlindedElement = 0370b0b4649c0880d44c421a3ca7c915b1b6ffa61f5a1290aa2
2258b006d148e5c105d47725e1ee1b2483b9c5666384038
EvaluationElement = 036d0aaf31ec411ef8e11c68551434883468e56cbd5d615a
c8c52b9dc7af326889d52d7466c5eed47f8c89707976aadc64
Output = ca7dc32dc6434101f35a790717dd591e5963acc86d20fda68011fe228fb
76be8da7f42c6a92284df88fb8e69480a3cb9
```

A.4.2. VOPRF Mode

```
Seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3
KeyInfo = 74657374206b6579
skSm = 514fb6fe2e66af1383840759d56f71730331280f062930ee2a2f7ea42f935
acf94087355699d788abfdf09d19a5c85ac
pkSm = 02f773b99e65ad26e8cd20614910ce7ad74c1baa5bdbfd9f124389dc8ef44
b5989f5bf036f6802dc2242fd7068b73da29f
```

A.4.2.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = 504650f53df8f16f6861633388936ea23338fa65ec36e0290022b48eb562
889d89dbfa691d1cde91517fa222ed7ad364
BlindedElement = 03022e23d8356d74d8f9a24ade759fb4e7cf050d1a770110878
83d4db52f16751d8d987fa49764c157c1039c4cdfa5ef7a
EvaluationElement = 0202bdefbc2d55a37aa848df5efc561055235d9190da9ec3
0ccfb84d93b033a29c4fb1968c55c63a0b90a205e1e9c4c19f
Proof = 929ee0254047350f580cdbd6fca706a9d110e4fc0aa1383af8d35a536795
69c038d90900e8810eca177b9cfd6a2d0f1fb5ed7a2e0f3107719cbd9c74ab7d9502
79869f67551b629c3706c8f9cee651d700453ca44e43b0a08c05502cd28f3960
ProofRandomScalar = 803d955f0e073a04aa5d92b3fb739f56f9db001266677f62
c095021db018cd8cbb55941d4073698ce45c405d1348b7b1
Output = 7eb3cc88d920431c3a5ea3fb6e36b515b6d82c5ef537e285918fe7c741e
97819ce029657d6cced0f8850f47ff281c444
```

A.4.2.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 504650f53df8f16f6861633388936ea23338fa65ec36e0290022b48eb562
889d89dbfa691d1cde91517fa222ed7ad364
BlindedElement = 037ae30a62126a39ca791aadafb65769c812a559c7da92820e1
43350b6bb8cefb543af2e0179664f9cd0d1499c018a0b18
EvaluationElement = 0355f95a68e8c4f0d40910e9a85f09109e4e7fff84f75db1
a4aa8e21c451ac2d872113b497bea6c0be1b535241557032a2
Proof = f4ec262642fc9981fe5d1f0a3737f2d09ec9b056f577224013f5a3d09812
fb22c6b45e17150d8fe3a8c7e63094cdf40a60ae1e50fc2e1678954c1ecbaed2f7d0
7e6d597fffedc7aca450ed64164c46e62d1326ff1f6eaeba4b5dd151e953e060
ProofRandomScalar = 803d955f0e073a04aa5d92b3fb739f56f9db001266677f62
c095021db018cd8cbb55941d4073698ce45c405d1348b7b1
Output = fb538f84dae5f214c5adfcf529c6fe63bc46d6a4073d540cf0dabcc7c8e
0f3c1b43b606002a9aa52ae158a19d900c136
```

A.4.2.3. Test Vector 3, Batch Size 2

```
Input = 00,5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 504650f53df8f16f6861633388936ea23338fa65ec36e0290022b48eb562
889d89dbfa691d1cde91517fa222ed7ad364,803d955f0e073a04aa5d92b3fb739f5
6f9db001266677f62c095021db018cd8cbb55941d4073698ce45c405d1348b7b1
BlindedElement = 03022e23d8356d74d8f9a24ade759fb4e7cf050d1a770110878
83d4db52f16751d8d987fa49764c157c1039c4cdfa5ef7a,031ee43111a2406b09eb
4fb2a3a5fd7c690c0aa51158af766c9df1428bb18195f054c5f68ae1863e6ab3dd42
98b3db712b
EvaluationElement = 0202bdefbc2d55a37aa848df5efc561055235d9190da9ec3
0ccfb84d93b033a29c4fb1968c55c63a0b90a205e1e9c4c19f,021fdbb3b92cf4f8e
04534bc1a9f62596667c3ea49a6e89f1610b9f7f89708e8730df159827ea92e26fcf
db2063920c89c
Proof = 9cc7fe5a120cec6ef0d877260cf1af1861f281aa0015f371c8830f93f286
8f5891ee6f32ec6fcbe130a50de24c93b131261eb4a242941c8d5ad9ad2f2be402d9
386ac4afcf5e5498f35cc3db0442a77e139eb56a7b3435177e7bf1a48cef184a
ProofRandomScalar = a097e722ed2427de86966910acba9f5c350e8040f828bf6c
eca27405420cdf3d63cb3aef005f40ba51943c8026877963
Output = 7eb3cc88d920431c3a5ea3fb6e36b515b6d82c5ef537e285918fe7c741e
97819ce029657d6cced0f8850f47ff281c444,fb538f84dae5f214c5adfcf529c6fe
63bc46d6a4073d540cf0dabcc7c8e0f3c1b43b606002a9aa52ae158a19d900c136
```

A.4.3. POPRF Mode

```
Seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3
KeyInfo = 74657374206b6579
skSm = 0fcba4a204f67d6c13f780e613915f755319aaa3cb03cd20a5a4a6c403a48
12a4fff5d3223e2c309aa66b05cb7611fd4
pkSm = 03a571100213c4356177af14a7039cfee270ad1f9abde42ac3418c501209e
d7b2fc0d4aa3373c12ba956fb555b02843fc8
```

A.4.3.1. Test Vector 1, Batch Size 1

```
Input = 00
Info = 7465737420696e666f
Blind = 504650f53df8f16f6861633388936ea23338fa65ec36e0290022b48eb562
889d89dbfa691d1cde91517fa222ed7ad364
BlindedElement = 03156aece0ce92e9eb8f7a9b7f6bd30230a048d41384f2fe49f
1f9f69e180c23390e3ba8d0ee66dde6d637f03c06385f76
EvaluationElement = 02352ec7586660cc4257a9e78366727341db0825e431fc82
4a70a91019b67be26d8b880b2d4d8e734207d4a21a23429d74
Proof = 77bb1ca3ba4013b93ccb302db838839098eca743de542d3c79d189f2adf0
01999583a01aead6c248a32ff13b7f1f3d6b2dd04f653a5beb0f0394ad83ce5e79ea
08ae029d669b918b6d62ed3b77b08a07f04bbc341fae06444d196746da4da884
ProofRandomScalar = 803d955f0e073a04aa5d92b3fb739f56f9db001266677f62
c095021db018cd8cbb55941d4073698ce45c405d1348b7b1
Output = fa15c0fe8706ac256dfd3c38d21ba0cd57b927cfcf3e4d6d5554ec1272e
670079b95cdbb2778e0df22baf50f33e12607
```

A.4.3.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Info = 7465737420696e666f
Blind = 504650f53df8f16f6861633388936ea23338fa65ec36e0290022b48eb562
889d89dbfa691d1cde91517fa222ed7ad364
BlindedElement = 02d46e0e2d27d8bb126e1201e881d0070b8807cb5635687b20d
d4a3a248e7a40c50a1ad3e905e43342771eb23bc8827a00
EvaluationElement = 030879805ff65cb536293a1449c00824e55c4c1b25379f2e
c17d97923055169a6d97b46ed7b11bb661cc8cb9535abc3d66
Proof = 9982a8501f45839213441d4ec501cf496d06fffab65f13ca3b3e66d21398
fe9e0e04aafdf50eae214fa9cccad3c53d524d0f8c185ed60b11fcf5c7e82e10a8d3
f3b2ce1e4a004d65e6ad596eeb5738453465d881f2770858cd46ac32f0e16121
ProofRandomScalar = 803d955f0e073a04aa5d92b3fb739f56f9db001266677f62
c095021db018cd8cbb55941d4073698ce45c405d1348b7b1
Output = 77cb533216c32cac017d706d5f0ee4630bcb0bfefbb980d95e98dc240ab
c70a944a44cde69b805aee3a39b2eb7d834be
```

A.4.3.3. Test Vector 3, Batch Size 2

```
Input = 00,5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Info = 7465737420696e666f
Blind = 504650f53df8f16f6861633388936ea23338fa65ec36e0290022b48eb562
889d89dbfa691d1cde91517fa222ed7ad364,803d955f0e073a04aa5d92b3fb739f5
6f9db001266677f62c095021db018cd8cbb55941d4073698ce45c405d1348b7b1
BlindedElement = 03156aece0ce92e9eb8f7a9b7f6bd30230a048d41384f2fe49f
1f9f69e180c23390e3ba8d0ee66dde6d637f03c06385f76,025663d73e3418039fdd
ea1a212d254ec0103f28904e588b73c7da8298347706b2f69902a98e8d01c7aaa69a
297b14c7dc
EvaluationElement = 02352ec7586660cc4257a9e78366727341db0825e431fc82
4a70a91019b67be26d8b880b2d4d8e734207d4a21a23429d74,02f8e532fabdd09bb
2a7391a2a80c14f265c0456009199b77eefac1013d4a4f449dfe46d5d6d2d4d74f8c
9fb1e2868b611
Proof = f8c938b5d2aff7d1a05ecdcf4178d682fe7b35c375be5db88dfa59f488c6
e4a68d4f99f16330a06f918e264ad68a78fdfad91446b72e1a3da2a65e531d520dd0
4fd91dd49b09037648e04a44e83d0dfd2aab7627e7389818924ad9bff591d646
ProofRandomScalar = a097e722ed2427de86966910acba9f5c350e8040f828bf6c
eca27405420cdf3d63cb3aef005f40ba51943c8026877963
Output = fa15c0fe8706ac256dfd3c38d21ba0cd57b927cfcf3e4d6d5554ec1272e
670079b95cdbb2778e0df22baf50f33e12607,77cb533216c32cac017d706d5f0ee4
630bcb0bfefbb980d95e98dc240abc70a944a44cde69b805aee3a39b2eb7d834be
```

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

A.5.1. OPRF Mode

```
Seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3
KeyInfo = 74657374206b6579
skSm = 0152e55f3a5d836ab6c2091a904ba4b4f92e51ba59ecc211b4fc771f7c6c8
b17fcbbb2bed8a65afd7811ceeec3eac83df6a58515b6d3c71ee0ffc349e28c3fb78
d83
```

A.5.1.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = 00d1dccf7a51bafaf75d4a866d53d8cafe4d504650f53df8f16f68616333
88936ea23338fa65ec36e0290022b48eb562889d89dbfa691d1cde91517fa222ed7a
d364
BlindedElement = 03016480f33f005c8a8eb1003e48ebc22e082d0b86678f8460e
df21cc1518a13bfc0001fa143d474b18214188d93a7b3124b1b385db4cd4e356ad24
923ae55d70ce8a7
EvaluationElement = 03005fdb56bf49fcd073b1c4cfb42ceef5666c709785ae82
d659e4d75c0f5591cbf812ca9ffd992ac67c1877b63978f417687a2a6c17697e858c
f715843f9e4235566a
Output = ddcaaceceec790f4858a09f3e06e74e8b0841681a3d45ab1393d0948379
43f782d9ed22ae716a642d4ee428ddf1dae9ff631047864b99a305412aceb7efafa3
2
```

A.5.1.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 00d1dccf7a51bafaf75d4a866d53d8cafe4d504650f53df8f16f68616333
88936ea23338fa65ec36e0290022b48eb562889d89dbfa691d1cde91517fa222ed7a
d364
BlindedElement = 02000e860d3b8205e0cb4f289771c8e6189b47c60cbff24459e
12a60317ac242e9cb36ab033a620cdee5628ecae4a81303e7464d52194d801756911
fd7ddfa5430e69c
EvaluationElement = 0300e2663f17144682b25de378531abd6d065b770eec073a
42494719f27748f75b4ab11aecb06bf8815bcc9eeb3ce54978605bd8a54c22a1dea6
2da1ae5f9f5e5e90f4
Output = 287712c6dbed773f39925fec0ad686dfda4a679cc7e88fa60ba9d3a7d71
2a11d4a0445995391ba56cfb018922e0d4bb4b25ec0965a33170c9b00f45c361b021
5
```

A.5.2. VOPRF Mode

```
Seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3
3a3
KeyInfo = 74657374206b6579
skSm = 00fb5507f94782c5b72acc16b9eb21064f86b4aa525b9865258d157b0431a
b5c3515fc975fa19ddb28129c969992b31d8946c4e354bc49458bb25fae58f10ac3f
678
pkSm = 0301322c63ad53e079791739169e011f362f4396a8e93fceeee9cd814d471
80e75ffd717820fe9e9c763fa595340cd80989c31fbd0200572080752c73b80b7532
2f300
```

A.5.2.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = 00d1dccf7a51bafaf75d4a866d53d8cafe4d504650f53df8f16f68616333
88936ea23338fa65ec36e0290022b48eb562889d89dbfa691d1cde91517fa222ed7a
d364
BlindedElement = 02016dafe8eee47b591592705ce4d5231563b637e5a51b425b8
81f1cc576c53caae4ec59fd6e3a918d5c35e6db77cf3a5862b71a8b6c7eaded3ebdf
0c6e14778c03a8c
EvaluationElement = 020124a0ee09ade261bbf67e1e3d296655c97e6c5c14c71a
386e636d8f55d29f5f6dcec954ff28bfc7e6e63240a52bf278ae94b312be3d8bf850
55d2a1dbab687905b0
Proof = 00156561564a9128de6e2fb92d0ee065bb19192ff86549c37fab777f2d57
a951ff94b3832162cf02ad73287a0f0906045878105d8ab54a7cc9a1a0039d0cb241
ebd10197e5cef77e8fbe0414f86b86fe2e823e0d8dbdcf2ccac54d273e814da062ba
941a27d1e7e28c44cdbdaffe392cc915bf8b9add15d51b68afd6e88a52d07ff8b3d1
ProofRandomScalar = 015e80ae32363b32cb76ad4b95a5a34e46bb803d955f0e07
3a04aa5d92b3fb739f56f9db001266677f62c095021db018cd8cbb55941d4073698c
e45c405d1348b7b1
Output = 16a9387153bf7fa2c733d42f299877324cfce3b39093e72067c3d59948b
f745d77b2fe9180ffb442ec45b575eb4108d2b6f207cbfabd7bc540ad2a087cfabca
2
```

A.5.2.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 00d1dccf7a51bafaf75d4a866d53d8cafe4d504650f53df8f16f68616333
88936ea23338fa65ec36e0290022b48eb562889d89dbfa691d1cde91517fa222ed7a
d364
BlindedElement = 02008f585341e32244d67033ddcf4c1cc30f7661c4cfc177f09
82c69bf9c90e1da02d86a26ece60b8c42b278a1dc85afcc9cbc6aedff15cc092af03
5100b915c2bb4df
EvaluationElement = 03006cfeb22e141859e6a2050a714bde8ab8109abb2b42bc
8f18ace67121c1811c9e95e7cf8ffd4f13f8cee80fc3c69318b0eb30ecdf6e7d7e84
faefa6f0b8299217fe
Proof = 01db7070ab756e8c2b12cb81c40daac6ef1d5137be3626a10ee867b0b736
ae5ab05aadbc3ee3d1d0202b7687e1614765893cba67b307c67a8a4ce7b3eaf3ba64
204901ce6f8dc9234d27373b1027982d7e3bb196d157403f50c2f1bf0fa701753ef6
3d7265c0b1016e662456d4bdea55b3d983350b2c2ce80e192897161a1b780046b952
ProofRandomScalar = 015e80ae32363b32cb76ad4b95a5a34e46bb803d955f0e07
3a04aa5d92b3fb739f56f9db001266677f62c095021db018cd8cbb55941d4073698c
e45c405d1348b7b1
Output = 0163635204be5347419796f3564b36d6e89c9170e4fcca5b6df79d3f676
f641b2ae3ae1a64cc49f3d788e276abe14e3c38bb2f92fdba0b45ed122a6930e7d96
1
```

A.5.2.3. Test Vector 3, Batch Size 2

```
Input = 00,5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 00d1dccf7a51bafaf75d4a866d53d8cafe4d504650f53df8f16f68616333
88936ea23338fa65ec36e0290022b48eb562889d89dbfa691d1cde91517fa222ed7a
d364,015e80ae32363b32cb76ad4b95a5a34e46bb803d955f0e073a04aa5d92b3fb7
39f56f9db001266677f62c095021db018cd8cbb55941d4073698ce45c405d1348b7b
1
BlindedElement = 02016dafe8eee47b591592705ce4d5231563b637e5a51b425b8
81f1cc576c53caae4ec59fd6e3a918d5c35e6db77cf3a5862b71a8b6c7eaded3ebdf
0c6e14778c03a8c,03005467c05309dd2b9ef584dd33ae30e93ae5508f2ceda71497
63b4b44fe797f7d0f4c7441298a0ed821ede9ebdc8c0215f96db57c64feb734a145f
00d00f0f222db1
EvaluationElement = 020124a0ee09ade261bbf67e1e3d296655c97e6c5c14c71a
386e636d8f55d29f5f6dcec954ff28bfc7e6e63240a52bf278ae94b312be3d8bf850
55d2a1dbab687905b0,0300fdf99a9eb28097074daf75ba9fe16868690b16165f58f
9c4fa266d5fffa5a87026a98ac3b0ca6dc7e42f49140a004c325646aec5ddc778db7
08748cc2f632ed937
Proof = 01935896f4c03ea5257d6471677f191ea7dfc777cc1e15f82e423cf1948c
440ee56a1c5a8627aad8da8e507a7f382b45255e55a1f1afc99c6b14237ce7cf0855
40fa000fe413be351bd11ac910b1d4af34d2c97c7b7a53438340dd659272f3d86470
35b13cd8072903b9a3adf8e89bfb1f77d732fa224f32674506e3e88e29ce182186e3
ProofRandomScalar = 01ec21c7bb69b0734cb48dfd68433dd93b0fa097e722ed24
27de86966910acba9f5c350e8040f828bf6ceca27405420cdf3d63cb3aef005f40ba
51943c8026877963
Output = 16a9387153bf7fa2c733d42f299877324cfce3b39093e72067c3d59948b
f745d77b2fe9180ffb442ec45b575eb4108d2b6f207cbfabd7bc540ad2a087cfabca
2,0163635204be5347419796f3564b36d6e89c9170e4fcca5b6df79d3f676f641b2a
e3ae1a64cc49f3d788e276abe14e3c38bb2f92fdba0b45ed122a6930e7d961
```

A.5.3. POPRF Mode

```
Seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a
3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3
3a3
KeyInfo = 74657374206b6579
skSm = 01e0993daeb97f8fc8176089e4e6adb4c03dc9b18daf7e976ed7fa6f3cb89
c40c6a84156f20371ef23bfe6e049423244d7d746c79ad380ac7fe285aba162419e9
012
pkSm = 0301264d23f5d1d615f9747d2a7177a419dabde6ca0f5a047979dbe9bce33
7241b7d2959025476f354c4f57017363d667b83b691fad8c172959963e6000de9533
f187a
```

A.5.3.1. Test Vector 1, Batch Size 1

```
Input = 00
Info = 7465737420696e666f
Blind = 00d1dccf7a51bafaf75d4a866d53d8cafe4d504650f53df8f16f68616333
88936ea23338fa65ec36e0290022b48eb562889d89dbfa691d1cde91517fa222ed7a
d364
BlindedElement = 0200e36b187060fef4f4cfef21cdb4ef8b5793a1bf44da95229
062303688d4cf6a50c16b7c943c79d91357223b56866351a17a9c7f49730fd28add9
301d399c0cf206c
EvaluationElement = 03014e216c05cf1d108829946891cc44693b0a411851a03f
c439130054d920eb8ad596a4dfa5314f68d298a094777855aa55c98480575a3816cf
ac52f838693e0e7fe5
Proof = 00c5a46ff1e7d8cd2711daf8ec8752451c4c7ed815f3e8d51db64f1eed83
a7cc33f0f99ce067676c478bd616a9ef6377994e4bd69051424a576a4e26f0ec7ed8
1fd000b7ae1eaee9e5b6991afdbb2c9c29a04e2ab3a2066df89308410a59267a60a2
2a47666de009646c78e9094c9f4de177a620e97f63e35ada0c8b438b4605248c9087
ProofRandomScalar = 015e80ae32363b32cb76ad4b95a5a34e46bb803d955f0e07
3a04aa5d92b3fb739f56f9db001266677f62c095021db018cd8cbb55941d4073698c
e45c405d1348b7b1
Output = 3be90ca19fbe2fc250de62792c7cf4b6b5555c8655fce1694fc7563d5d4
c5001efd1e91fbbaea31d75e33dbdefe57420c395f1ac805cc0095c4d81a0beddcb0
1
```

A.5.3.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Info = 7465737420696e666f
Blind = 00d1dccf7a51bafaf75d4a866d53d8cafe4d504650f53df8f16f68616333
88936ea23338fa65ec36e0290022b48eb562889d89dbfa691d1cde91517fa222ed7a
d364
BlindedElement = 0300357933cc17cdcce862b794a4161d8eb10d23009695639e3
fdc8dffc235e19e92e0a3d3c7c6249dd9dcd02da0a8f061d89b6809d3292951ee0e9
ead21a62d1335fe
EvaluationElement = 0300a5132ae9c429dd33b25c051f45451c6e54e154d698c3
f3d8820bd9607e7a65762911c647b3460be166f37ba443bf000b23552298f14e0555
b3f0ddf0e900e1d38c
Proof = 0004f0791cbe6ac6f4074834e172beedea19ecd3a2c504a71fd870b42314
d3b072633a8265c774668274dcbcaebf1726768fab4edec69a33a7d37095ebef3e1b
b44900f0a175b56ceeae8a87bc5553405e0b030ebcf8303befc5890c8afa1e61fd41
66480ff428eae4193f12bbf1fc31d5d7196ce8692e37bc9a63cdf4c9fafe10a2dc9a
ProofRandomScalar = 015e80ae32363b32cb76ad4b95a5a34e46bb803d955f0e07
3a04aa5d92b3fb739f56f9db001266677f62c095021db018cd8cbb55941d4073698c
e45c405d1348b7b1
Output = 1d90446522e3c131e90be2e4f372959ae5ab4f25ca98e83e5e62d6336c4
8b5ec22fc6083d2b050cad2bbc22ae7115c2b934d965ffe74aaa43c905cd2af76728
d
```

A.5.3.3. Test Vector 3, Batch Size 2

```
Input = 00,5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Info = 7465737420696e666f
Blind = 00d1dccf7a51bafaf75d4a866d53d8cafe4d504650f53df8f16f68616333
88936ea23338fa65ec36e0290022b48eb562889d89dbfa691d1cde91517fa222ed7a
d364,015e80ae32363b32cb76ad4b95a5a34e46bb803d955f0e073a04aa5d92b3fb7
39f56f9db001266677f62c095021db018cd8cbb55941d4073698ce45c405d1348b7b
1
BlindedElement = 0200e36b187060fef4f4cfef21cdb4ef8b5793a1bf44da95229
062303688d4cf6a50c16b7c943c79d91357223b56866351a17a9c7f49730fd28add9
301d399c0cf206c,03007530916e8ec76199429667a82ca4df65b913d8b1fb157319
e73706f118b4f46047c01b7da024bdf5a06f2f4e879b1a1cd3fcb1ca2c37ce158cc8
625e76b3bb1cc4
EvaluationElement = 03014e216c05cf1d108829946891cc44693b0a411851a03f
c439130054d920eb8ad596a4dfa5314f68d298a094777855aa55c98480575a3816cf
ac52f838693e0e7fe5,0200005cf5e719b3066dcf0fbd6228bc921cebccc49feb1ac
be9d9c4c88f4169e1d0d5408f92ad9f599c2f5f6d7d4c6e575e86f64c4eead2bb9b3
e8e04d141a90b7382
Proof = 00d846f4a2a7722fe6a24e7257e43d88c3e01977282fba352c08fd38b69b
f1df64f90660b03b73abba50cb389af3d602da66411401d3c9f87bcb6363d6406e0a
cad3018a44bcda83524d4a48f0ed96ebca96d7626b634ba28fcba0c21956fc90c516
859df8ba6edeb7a44daeeec51c3a56b79c1f9e211e9974e5f293ade221523953d12f
ProofRandomScalar = 01ec21c7bb69b0734cb48dfd68433dd93b0fa097e722ed24
27de86966910acba9f5c350e8040f828bf6ceca27405420cdf3d63cb3aef005f40ba
51943c8026877963
Output = 3be90ca19fbe2fc250de62792c7cf4b6b5555c8655fce1694fc7563d5d4
c5001efd1e91fbbaea31d75e33dbdefe57420c395f1ac805cc0095c4d81a0beddcb0
1,1d90446522e3c131e90be2e4f372959ae5ab4f25ca98e83e5e62d6336c48b5ec22
fc6083d2b050cad2bbc22ae7115c2b934d965ffe74aaa43c905cd2af76728d
```

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