

Workgroup: Network Working Group  
Internet-Draft: draft-irtf-cfrg-voprf-19  
Published: 24 January 2023  
Intended Status: Informational  
Expires: 28 July 2023

A            A. Davidson            A. Faz-Hernandez        N. Sullivan  
              uBrave Software        Cloudflare, Inc.        Cloudflare, Inc.  
              t  
              h  
              o  
              r  
              s  
              :  
              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. This document is a product of the Crypto Forum Research Group (CFRG) in the IRTF.

### **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 28 July 2023.

## Copyright Notice

Copyright (c) 2023 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](#)
    - 3.2.1. [Deterministic Key Generation](#)
  - 3.3. [Online Protocol](#)
    - 3.3.1. [OPRF Protocol](#)
    - 3.3.2. [VOPRF Protocol](#)
    - 3.3.3. [POPRF Protocol](#)
4. [Ciphersuites](#)
  - 4.1. [OPRF\(ristretto255, SHA-512\)](#)
  - 4.2. [OPRF\(decaf448, SHAKE-256\)](#)
  - 4.3. [OPRF\(P-256, SHA-256\)](#)
  - 4.4. [OPRF\(P-384, SHA-384\)](#)
  - 4.5. [OPRF\(P-521, SHA-512\)](#)
  - 4.6. [Future Ciphersuites](#)
  - 4.7. [Random Scalar Generation](#)
    - 4.7.1. [Rejection Sampling](#)
    - 4.7.2. [Random Number Generation Using Extra Random Bits](#)
5. [Application Considerations](#)
  - 5.1. [Input Limits](#)
  - 5.2. [External Interface Recommendations](#)
  - 5.3. [Error Considerations](#)
  - 5.4. [POPRF Public Input](#)
6. [IANA considerations](#)
7. [Security Considerations](#)
  - 7.1. [Security Properties](#)
  - 7.2. [Security Assumptions](#)
    - 7.2.1. [OPRF and VOPRF Assumptions](#)
    - 7.2.2. [POPRF Assumptions](#)
    - 7.2.3. [Static Diffie Hellman Attack and Security Limits](#)

- [7.3. Domain Separation](#)
- [7.4. Timing Leaks](#)
- [8. Acknowledgements](#)
- [9. References](#)
  - [9.1. Normative References](#)
  - [9.2. Informative References](#)
- [Appendix A. Test Vectors](#)
  - [A.1. ristretto255-SHA512](#)
    - [A.1.1. OPRF Mode](#)
    - [A.1.2. VOPRF Mode](#)
    - [A.1.3. POPRF Mode](#)
  - [A.2. decaf448-SHAKE256](#)
    - [A.2.1. OPRF Mode](#)
    - [A.2.2. VOPRF Mode](#)
    - [A.2.3. POPRF Mode](#)
  - [A.3. P256-SHA256](#)
    - [A.3.1. OPRF Mode](#)
    - [A.3.2. VOPRF Mode](#)
    - [A.3.3. POPRF Mode](#)
  - [A.4. P384-SHA384](#)
    - [A.4.1. OPRF Mode](#)
    - [A.4.2. VOPRF Mode](#)
    - [A.4.3. POPRF Mode](#)
  - [A.5. P521-SHA512](#)
    - [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 [[OPAQUE](#)]. Verifiable OPRFs are necessary in some applications such as Privacy Pass [[PRIVACYPASS](#)]. 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.

This document represents the consensus of the Crypto Forum Research Group (CFRG). It is not an IETF product and is not a standard.

## 1.1. Change log

### [draft-19](#):

- \*Fix error

### [draft-18](#):

- \*Apply editorial suggestions from CFRG chair review.

### [draft-17](#):

- \*Change how suites are identified and finalize test vectors.

- \*Apply editorial suggestions from IRTF chair review.

### [draft-16](#):

- \*Apply editorial suggestions from document shepherd.

### [draft-15](#):

- \*Apply editorial suggestions from CFRG RGLC.

### [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  $\text{len}(x)$  to denote its length in bytes.

\*For two byte arrays  $x$  and  $y$ , write  $x \parallel y$  to denote their concatenation.

\* $\text{I2OSP}(x, \text{xLen})$ : Converts a non-negative integer  $x$  into a byte array of specified length  $\text{xLen}$  as described in [[RFC8017](#)]. Note that this function returns a byte array in big-endian byte order.

\*The notation  $T \text{ U}[N]$  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  $0 \leq j < N$ .

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  $2^{16} - 1$  bytes. This length restriction exists because `PublicInput` and `PrivateInput` values are length-prefixed with two bytes before use throughout the protocol.

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 using 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  $N_h$  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  $G$  for performing all mathematical operations. In prime-order groups, any element (other than the identity) 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 by  $r$  is equivalent to the repeated application of the group operation on an element  $A$  with itself  $r-1$  times, this is denoted as  $r*A = A + \dots + A$ . For any element  $A$ ,  $p*A=I$ . The case when the scalar multiplication is performed on the group generator is denoted as  $\text{ScalarMultGen}(r)$ . 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  $\text{GF}(p)$ , a prime field of order  $p$ , and are represented as the set of integers defined by  $\{0, 1, \dots, p-1\}$ . 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): Deterministically maps an array of bytes  $x$  to an element of Group. The map must ensure that, for any adversary receiving  $R = \text{HashToGroup}(x)$ , it is computationally difficult to reverse the mapping. This function is optionally parameterized by a domain separation tag (DST); see [Section 4](#). Security properties of this function are described in [[I-D.irtf-cfrg-hash-to-curve](#)].

\*HashToScalar(x): Deterministically maps an array of bytes  $x$  to an element in  $\text{GF}(p)$ . This function is optionally parameterized by a DST; see [Section 4](#). Security properties of this function are described in [[I-D.irtf-cfrg-hash-to-curve](#)], [Section 10.5](#).

\*RandomScalar(): Chooses at random a non-zero element in  $\text{GF}(p)$ .

\*ScalarInverse(s): Returns the inverse of input Scalar  $s$  on  $\text{GF}(p)$ .

\*SerializeElement(A): Maps an Element  $A$  to a canonical byte array buf of fixed length  $N_e$ .

\*DeserializeElement(buf): Attempts to map a byte array buf to an Element  $A$ , and fails if the input is not the valid canonical byte representation of an element of the group. This function can raise a DeserializeError if deserialization fails or  $A$  is the identity element of the group; see [Section 4](#) for group-specific input validation steps.

\*SerializeScalar(s): Maps a Scalar  $s$  to a canonical byte array buf of fixed length  $N_s$ .

\*DeserializeScalar(buf): Attempts to map a byte array buf to a Scalar  $s$ . This function can raise a DeserializeError if deserialization fails; see [Section 4](#) for group-specific input validation steps.

[Section 4](#) contains details for the implementation of this interface for different prime-order groups instantiated over elliptic curves. In particular, for some choices of elliptic curves, e.g., those detailed in [[RFC7748](#)], which require accounting for cofactors, [Section 4](#) describes required steps necessary to ensure the resulting group is of prime order.

## 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), which is used in the construction of VOPRF and POPRF. 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  $k \cdot A == B$  and  $k \cdot C[i] == D[i]$  for each  $i$  in  $[0, \dots, m - 1]$ . 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 to as the OPRF, VOPRF, and POPRF modes with the following properties.

In the OPRF mode, a client and server interact to compute output =  $F(\text{skS}, \text{input})$ , 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.

```

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

                                blindedElement
                                ----->

                                evaluatedElement = BlindEvaluate(skS, blindedElement)

                                evaluatedElement
                                <-----

output = Finalize(input, blind, evaluatedElement)

```

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.

```

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

                                blindedElement
                                ----->

                                evaluatedElement, proof = BlindEvaluate(skS, pkS,
                                                                           blindedElement)

                                evaluatedElement, proof
                                <-----

output = Finalize(input, blind, evaluatedElement,
                  blindedElement, pkS, proof)

```

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.

```

Client(input, pkS, info) <---- pkS ----- Server(skS, pkS, info)
-----
blind, blindedElement, tweakedKey = Blind(input, info, pkS)

blindedElement
----->

evaluatedElement, proof = BlindEvaluate(skS, blindedElement,
                                         info)

evaluatedElement, proof
<-----

output = Finalize(input, blind, evaluatedElement,
                  blindedElement, proof, info, tweakedKey)

```

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  
protocol variants.

Additionally, each protocol variant is instantiated with a ciphersuite, or suite. Each ciphersuite is identified with an ASCII string identifier, referred to as identifier; see [Section 4](#) for the set of initial ciphersuite values.

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

```

def CreateContextString(mode, identifier):
    return "OPRFV1-" || I2OSP(mode, 1) || "-" || identifier

```

### 3.2. Key Generation and Context Setup

In the offline setup phase, the server generates a fresh, random key pair (skS, pkS). There are two ways to generate this key pair. The first of which is using the GenerateKeyPair function described below.



Input: None

Output:

Scalar skS  
Element pkS

Parameters:

Group G

```
def GenerateKeyPair():  
    skS = G.RandomScalar()  
    pkS = G.ScalarMultGen(skS)  
    return skS, pkS
```

The second way to generate the key pair is via the deterministic key generation function `DeriveKeyPair` described in [Section 3.2.1](#). Applications and implementations can use either method in practice.

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 identifier. 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(identifier, skS):  
    contextString = CreateContextString(modeOPRF, identifier)  
    return OPRFServerContext(contextString, skS)  
  
def SetupOPRFClient(identifier):  
    contextString = CreateContextString(modeOPRF, identifier)  
    return OPRFClientContext(contextString)
```

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

```
def SetupVOPRFServer(identifier, skS):  
    contextString = CreateContextString(modeVOPRF, identifier)  
    return VOPRFServerContext(contextString, skS)  
  
def SetupVOPRFClient(identifier, pkS):  
    contextString = CreateContextString(modeVOPRF, identifier)  
    return VOPRFClientContext(contextString, pkS)
```

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

```
def SetupPOPRFServer(identifier, skS):  
    contextString = CreateContextString(modePOPRF, identifier)  
    return POPRFServerContext(contextString, skS)  
  
def SetupPOPRFClient(identifier, pkS):  
    contextString = CreateContextString(modePOPRF, identifier)  
    return POPRFClientContext(contextString, pkS)
```

### 3.2.1. Deterministic Key Generation

This section describes a deterministic key generation function, `DeriveKeyPair`. It accepts a seed of  $N_s$  bytes generated from a cryptographically secure random number generator and an optional (possibly empty) info string. The constant  $N_s$  corresponds to the size in bytes of a serialized Scalar and is defined in [Section 2.1](#). Note that by design knowledge of seed and info is necessary to compute this function, which means that the secrecy of the output private key ( $skS$ ) depends on the secrecy of seed (since the info string is public).

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
```

### 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 raising 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](#) for more information and requirements on input validation for each ciphersuite.

### 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)
```

An entity which knows both the secret key and the input can compute the PRF result 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, an entity which knows both the secret key and the input can compute the PRF result 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, an entity which knows both the secret key and the input can compute the PRF result 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.

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 7.2](#) for related discussion.

#### 4.1. OPRF(ristretto255, SHA-512)

This ciphersuite uses ristretto255 [[RISTRETTO](#)] for the Group and SHA-512 for the Hash function. The value of the ciphersuite identifier is "ristretto255-SHA512".

\*Group: ristretto255 [[RISTRETTO](#)]

-Order(): Return  $2^{252} + 2774231777372353535851937790883648493$  (see [[RISTRETTO](#)])

-Identity(): As defined in [[RISTRETTO](#)].

-Generator(): As defined in [[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().

-ScalarInverse(s): Returns the multiplicative inverse of input Scalar s mod Group.Order().

-RandomScalar(): Implemented by returning a uniformly random Scalar in the range  $[0, G.Order() - 1]$ . Refer to [Section 4.7](#) for implementation guidance.

-SerializeElement(A): Implemented using the 'Encode' function from Section 4.3.2 of [[RISTRETTO](#)]; Ne = 32.

-DeserializeElement(buf): Implemented using the 'Decode' function from Section 4.3.1 of [[RISTRETTO](#)]. Additionally, this function validates that the resulting element is not the group identity element. If these checks fail, deserialization returns an InputValidationError error.

-SerializeScalar(s): Implemented by outputting the little-endian 32-byte encoding of the Scalar value with the top three bits set to zero; Ns = 32.

-DeserializeScalar(buf): Implemented by attempting to deserialize a Scalar from a little-endian 32-byte string. This

function can fail if the input does not represent a Scalar in the range  $[0, G.Order() - 1]$ . Note that this means the top three bits of the input MUST be zero.

\*Hash: SHA-512; Nh = 64.

#### 4.2. OPRF(decaf448, SHAKE-256)

This ciphersuite uses decaf448 [[RISTRETTO](#)] for the Group and SHAKE-256 for the Hash function. The value of the ciphersuite identifier is "decaf448-SHAKE256".

\*Group: decaf448 [[RISTRETTO](#)]

-Order(): Return  $2^{446} - 13818066809895115352007386748515426880336692474882178609894547503885$

-Identity(): As defined in [[RISTRETTO](#)].

-Generator(): As defined in [[RISTRETTO](#)].

-RandomScalar(): Implemented by returning a uniformly random Scalar in the range  $[0, G.Order() - 1]$ . Refer to [Section 4.7](#) for implementation guidance.

-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()`.

-ScalarInverse(s): Returns the multiplicative inverse of input Scalar `s mod Group.Order()`.

-SerializeElement(A): Implemented using the 'Encode' function from Section 5.3.2 of [[RISTRETTO](#)]; `Ne = 56`.

-DeserializeElement(buf): Implemented using the 'Decode' function from Section 5.3.1 of [[RISTRETTO](#)]. Additionally, this function validates that the resulting element is not the group identity element. If these checks fail, deserialization returns an `InputValidationError` error.

-SerializeScalar(s): Implemented by outputting the little-endian 56-byte encoding of the Scalar value; `Ns = 56`.

-DeserializeScalar(buf): Implemented by attempting to deserialize a Scalar from a little-endian 56-byte string. This function can fail if the input does not represent a Scalar in the range  $[0, G.Order() - 1]$ .

\*Hash: SHAKE-256; Nh = 64.

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

This ciphersuite uses P-256 [[NISTCurves](#)] for the Group and SHA-256 for the Hash function. The value of the ciphersuite identifier is "P256-SHA256".

\*Group: P-256 (secp256r1) [[NISTCurves](#)]

-Order(): Return  
0xffffffff00000000ffffffffffffffffbce6faada7179e84f3b9cac2fc63  
2551.

-Identity(): As defined in [[NISTCurves](#)].

-Generator(): As defined in [[NISTCurves](#)].

-RandomScalar(): Implemented by returning a uniformly random Scalar in the range  $[0, G.Order() - 1]$ . Refer to [Section 4.7](#) for implementation guidance.

-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().

-ScalarInverse(s): Returns the multiplicative inverse of input  
Scalar  $s \bmod \text{Group.Order}()$ .

-SerializeElement(A): Implemented using the compressed  
Elliptic-Curve-Point-to-Octet-String method according to  
[\[SEC1\]](#);  $N_e = 33$ .

-DeserializeElement(buf): Implemented by attempting to  
deserialize a 33 byte input string to a public key using the  
compressed Octet-String-to-Elliptic-Curve-Point method  
according to [\[SEC1\]](#), and then performs partial public-key  
validation as defined in section 5.6.2.3.4 of [\[KEYAGREEMENT\]](#).  
This includes checking that the coordinates of the resulting  
point are in the correct range, that the point is on the  
curve, and that the point is not the group identity element.  
If these checks fail, deserialization returns an  
InputValidationError error.

-SerializeScalar(s): Implemented using the Field-Element-to-  
Octet-String conversion according to [\[SEC1\]](#);  $N_s = 32$ .

-DeserializeScalar(buf): Implemented by attempting to  
deserialize a Scalar from a 32-byte string using Octet-String-  
to-Field-Element from [\[SEC1\]](#). This function can fail if the  
input does not represent a Scalar in the range  $[0, G.Order() - 1]$ .

\*Hash: SHA-256;  $N_h = 32$ .

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

This ciphersuite uses P-384 [[NISTCurves](#)] for the Group and SHA-384 for the Hash function. The value of the ciphersuite identifier is "P384-SHA384".

\*Group: P-384 (secp384r1) [[NISTCurves](#)]

-Order(): Return  
0xfffc7634d81f437  
2ddf581a0db248b0a77aecec196accc52973.

-Identity(): As defined in [[NISTCurves](#)].

-Generator(): As defined in [[NISTCurves](#)].

-RandomScalar(): Implemented by returning a uniformly random Scalar in the range  $[0, G.Order() - 1]$ . Refer to [Section 4.7](#) for implementation guidance.

-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().

-ScalarInverse(s): Returns the multiplicative inverse of input  
Scalar  $s$  mod Group.Order().

-SerializeElement(A): Implemented using the compressed  
Elliptic-Curve-Point-to-Octet-String method according to  
[\[SEC1\]](#);  $N_e = 49$ .

-DeserializeElement(buf): Implemented by attempting to  
deserialize a 49-byte array to a public key using the  
compressed Octet-String-to-Elliptic-Curve-Point method  
according to [\[SEC1\]](#), and then performs partial public-key  
validation as defined in section 5.6.2.3.4 of [\[KEYAGREEMENT\]](#).  
This includes checking that the coordinates of the resulting  
point are in the correct range, that the point is on the  
curve, and that the point is not the point at infinity.  
Additionally, this function validates that the resulting  
element is not the group identity element. If these checks  
fail, deserialization returns an `InputValidationError` error.

-SerializeScalar(s): Implemented using the Field-Element-to-  
Octet-String conversion according to [\[SEC1\]](#);  $N_s = 48$ .

-DeserializeScalar(buf): Implemented by attempting to  
deserialize a Scalar from a 48-byte string using Octet-String-  
to-Field-Element from [\[SEC1\]](#). This function can fail if the  
input does not represent a Scalar in the range  $[0, G.Order() - 1]$ .

\*Hash: SHA-384;  $N_h = 48$ .

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

This ciphersuite uses P-521 [[NISTCurves](#)] for the Group and SHA-512 for the Hash function. The value of the ciphersuite identifier is "P521-SHA512".

\*Group: P-521 (secp521r1) [[NISTCurves](#)]

-Order(): Return

```
0x01ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffa51868783bf2f966b7fcc0148f709a5d03bb5c9b8899c47aebb6fb7
1e91386409.
```

-Identity(): As defined in [[NISTCurves](#)].

-Generator(): As defined in [[NISTCurves](#)].

-RandomScalar(): Implemented by returning a uniformly random Scalar in the range  $[0, G.Order() - 1]$ . Refer to [Section 4.7](#) for implementation guidance.

-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()`.

-ScalarInverse(s): Returns the multiplicative inverse of input Scalar `s` mod `Group.Order()`.

-SerializeElement(A): Implemented using the compressed Elliptic-Curve-Point-to-Octet-String method according to [[SEC1](#)]; `Ne = 67`.

-DeserializeElement(buf): Implemented by attempting to deserialize a 49 byte input string to a public key using the compressed Octet-String-to-Elliptic-Curve-Point method according to [[SEC1](#)], and then performs partial public-key validation as defined in section 5.6.2.3.4 of [[KEYAGREEMENT](#)]. This includes checking that the coordinates of the resulting point are in the correct range, that the point is on the curve, and that the point is not the point at infinity. Additionally, this function validates that the resulting element is not the group identity element. If these checks fail, deserialization returns an `InputValidationError` error.

-SerializeScalar(s): Implemented using the Field-Element-to-Octet-String conversion according to [[SEC1](#)]; `Ns = 66`.

-DeserializeScalar(buf): Implemented by attempting to deserialize a Scalar from a 66-byte string using `Octet-String-to-Field-Element` from [[SEC1](#)]. This function can fail if the input does not represent a Scalar in the range  $[0, G.Order() - 1]$ .

\*Hash: SHA-512; Nh = 64.

#### 4.6. 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 (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.

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](#). Future ciphersuites MUST describe how input validation is done for DeserializeElement and DeserializeScalar.

Additionally, future ciphersuites must take care when choosing the security level of the group. See [Section 7.2.3](#) for additional details.

#### 4.7. Random Scalar Generation

Two popular algorithms for generating a random integer uniformly distributed in the range  $[0, G.Order() - 1]$  are as follows:

##### 4.7.1. Rejection Sampling

Generate a random byte array with Ns bytes, and attempt to map to a Scalar by calling DeserializeScalar in constant time. If it succeeds, return the result. If it fails, try again with another random byte array, until the procedure succeeds. Failure to implement DeserializeScalar in constant time can leak information about the underlying corresponding Scalar.

As an optimization, if the group order is very close to a power of 2, it is acceptable to omit the rejection test completely. In particular, if the group order is p, and there is an integer b such that  $|p - 2^b|$  is less than  $2^{(b/2)}$ , then RandomScalar can simply return a uniformly random integer of at most b bits.

##### 4.7.2. Random Number Generation Using Extra Random Bits

Generate a random byte array with  $L = \text{ceil}(((3 * \text{ceil}(\log_2(G.Order())) / 2) / 8)$  bytes, and interpret it as an integer; reduce the integer modulo G.Order() and return the result. See [[I-D.irtf-cfrg-hash-to-curve](#)], [Section 5](#) for the underlying derivation of L.



## 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  $2^{16}-1$  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 OPFR 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 OPFR 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](#).

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

\*InvalidInputError: OPFR 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 7.2](#) 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. IANA considerations

This document has no IANA actions.

## 7. Security Considerations

This section discusses the security of the protocols defined in this specification, along with some suggestions and trade-offs that arise from the implementation of the protocol 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.

### 7.1. 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](#)].

Beyond client input secret, in the OPRF protocol, the server learns nothing about the output  $y$  of the function, nor does the client learn anything about the server's private key  $k$ .

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. Both client and server know the public input, but similar to the OPRF and VOPRF protocols, the server learns nothing about the client's private input or the output of the function, and the client learns 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. Note that partial obliviousness only applies to the POPRF variant because neither the OPRF nor VOPRF variants accept public input to the protocol.

Since the POPRF variant has a different syntax than the OPRF and VOPRF variants, i.e.,  $y = F(k, x, \text{info})$ , the pseudorandomness property is generalized:

\*Pseudorandomness: For a random sampling of  $k$ ,  $F$  is pseudorandom if the output  $y = F(k, x, \text{info})$  on any input pairs  $(x, \text{info})$  is indistinguishable from uniformly sampling any element in  $F$ 's range.

## 7.2. 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.

### 7.2.1. 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 the One-More Gap Computational Diffie Hellman assumption that is computationally difficult to solve in the corresponding prime-order group. In [\[JKK14\]](#), these properties are proven 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.

### 7.2.2. 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 batch, 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 Computational Diffie Hellman 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 Computational Diffie Hellman 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.](#))

### 7.2.3. 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 attacks, 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).

### 7.3. 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](#).

### 7.4. 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.

## 8. 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, Julia Hesse, Russ Housley, Kevin Lewi, Christopher Patton, and Bas Westerbaan also provided helpful input and contributions to the document.

## 9. References

### 9.1. Normative References

[I-D.irtf-cfrg-hash-to-curve] Faz-Hernandez, A., Scott, S., Sullivan, N., Wahby, R. S., and C. A. Wood, "Hashing to Elliptic Curves", Work in Progress, Internet-Draft, draft-irtf-cfrg-hash-to-curve-16, 15 June 2022, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hash-to-curve-16>>.

#### [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>>.

[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-decaf448-05, 29 November 2022, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-ristretto255-decaf448-05>>.

## 9.2. 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](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](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](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](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>>.

[NISTCurves] "Digital Signature Standard (DSS)", National Institute of Standards and Technology report, DOI 10.6028/nist.fips.186-4, July 2013, <<https://doi.org/10.6028/nist.fips.186-4>>.

[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>>.

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

[PRIVACYPASS] Celi, S., Davidson, A., Faz-Hernandez, A., Valdez, S., and C. A. Wood, "Privacy Pass Issuance Protocol", Work in Progress, Internet-Draft, draft-ietf-privacypass-protocol-07, 8 December 2022, <<https://datatracker.ietf.org/doc/html/draft-ietf-privacypass-protocol-07>>.

[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>>.

[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](https://doi.org/10.1007/978-3-031-07085-3_23)>.

## 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  $N_s$  bytes long.
- \*"BlindedElement": The blinded value output by Blind(), a serialized Element of  $N_e$  bytes long.
- \*"EvaluatedElement": The evaluated element output by BlindEvaluate(), a serialized Element of  $N_e$  bytes long.
- \*"Proof": The serialized Proof output from GenerateProof() composed of two serialized Scalar values each of  $N_s$  bytes long. Only present for VOPRF and POPRF test vectors.
- \*"ProofRandomScalar": The random scalar  $r$  computed in GenerateProof(), a serialized Scalar of  $N_s$  bytes long. Only present for VOPRF and POPRF test vectors.
- \*"Output": The protocol output, an opaque byte string of length  $N_h$  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,  $pk_{Sm}$  and  $sk_{Sm}$ , are listed under the mode for each ciphersuite. Both  $pk_{Sm}$  and  $sk_{Sm}$  are the serialized values of  $pk_S$  and  $sk_S$ , 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. ristretto255-SHA512

#### A.1.1. OPRF Mode

Seed = a3  
3a3  
KeyInfo = 74657374206b6579  
skSm = 5ebcea5ee37023ccb9fc2d2019f9d7737be85591ae8652ffa9ef0f4d37063  
b0e







**A.1.3.1. Test Vector 1, Batch Size 1**

Input = 00  
 Info = 7465737420696e666f  
 Blind = 64d37aed22a27f5191de1c1d69fad899d8862b58eb4220029e036ec4c1f6706  
 BlindedElement = c8713aa89241d6989ac142f22dba30596db635c772cbf25021fdd8f3d461f715  
 EvaluationElement = 1a4b860d808ff19624731e67b5eff20ceb2df3c3c03b906f5693e2078450d874  
 Proof = 41ad1a291aa02c80b0915fbfb0c0afa15a57e2970067a602ddb9e8fd6b7100de32e1ecff943a36f0b10e3dae6bd266cdeb8adf825d86ef27dbc6c0e30c52206  
 ProofRandomScalar = 222a5e897cf59db8145db8d16e597e8facb80ae7d4e26d9881aa6f61d645fc0e  
 Output = ca688351e88afb1d841fde4401c79efebb2eb75e7998fa9737bd5a82a152406d38bd29f680504e54fd4587eddcf2f37a2617ac2fbd2993f7bdf45442ace7d221

**A.1.3.2. Test Vector 2, Batch Size 1**

Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a  
 Info = 7465737420696e666f  
 Blind = 64d37aed22a27f5191de1c1d69fad899d8862b58eb4220029e036ec4c1f6706  
 BlindedElement = f0f0b209dd4d5f1844dac679acc7761b91a2e704879656cb7c201e82a99ab07d  
 EvaluationElement = 8c3c9d064c334c6991e99f286ea2301d1bde170b54003fb9c44c6d7bd6fc1540  
 Proof = 4c39992d55ffba38232cdac88fe583af8a85441fef7d7d1d4a8d0394cd1de77018bf135c174f20281b3341ab1f453fe72b0293a7398703384bed822bfdeec8908  
 ProofRandomScalar = 222a5e897cf59db8145db8d16e597e8facb80ae7d4e26d9881aa6f61d645fc0e  
 Output = 7c6557b276a137922a0bcfc2aa2b35dd78322bd500235eb6d6b6f91bc5b56a52de2d65612d503236b321f5d0bebc52b64b92e426f29c9b8b69f52de98ae507

**A.1.3.3. Test Vector 3, Batch Size 2**

Input = 00,5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a  
 Info = 7465737420696e666f  
 Blind = 64d37aed22a27f5191de1c1d69fad899d8862b58eb4220029e036ec4c1f6706,222a5e897cf59db8145db8d16e597e8facb80ae7d4e26d9881aa6f61d645fc0e  
 BlindedElement = c8713aa89241d6989ac142f22dba30596db635c772cbf25021fdd8f3d461f715,423a01c072e06eb1cce96d23acce06e1ea64a609d7ec9e9023f3049f2d64e50c  
 EvaluationElement = 1a4b860d808ff19624731e67b5eff20ceb2df3c3c03b906f5693e2078450d874,aa1f16e903841036e38075da8a46655c94fc92341887eb5819f46312adfc0504  
 Proof = 43fdb53be399cbd3561186ae480320caa2b9f36cca0e5b160c4a677b8bbf4301b28f12c36aa8e11e5a7ef551da0781e863a6dc8c0b2bf5a149c9e00621f02006  
 ProofRandomScalar = 419c4f4f5052c53c45f3da494d2b67b220d02118e0857cdbcf037f9ea84bbe0c  
 Output = ca688351e88afb1d841fde4401c79efebb2eb75e7998fa9737bd5a82a152406d38bd29f680504e54fd4587eddcf2f37a2617ac2fbd2993f7bdf45442ace7d221,7c6557b276a137922a0bcfc2aa2b35dd78322bd500235eb6d6b6f91bc5b56a52de2d65612d503236b321f5d0bebc52b64b92e426f29c9b8b69f52de98ae507









### A.3.1.1. Test Vector 1, Batch Size 1

Input = 00  
 Blind = 3338fa65ec36e0290022b48eb562889d89dbfa691d1cde91517fa222ed7a  
 d364  
 BlindedElement = 03723a1e5c09b8b9c18d1dcbca29e8007e95f14f4732d9346d4  
 90ffc195110368d  
 EvaluationElement = 030de02ffec47a1fd53efcdd1c6faf5bdc270912b8749e78  
 3c7ca75bb412958832  
 Output = a0b34de5fa4c5b6da07e72af73cc507cceeab48981b97b7285fc375345fe  
 495dd

### A.3.1.2. Test Vector 2, Batch Size 1

Input = 5a  
 Blind = 3338fa65ec36e0290022b48eb562889d89dbfa691d1cde91517fa222ed7a  
 d364  
 BlindedElement = 03cc1df781f1c2240a64d1c297b3f3d16262ef5d4cf10273488  
 2675c26231b0838  
 EvaluationElement = 03a0395fe3828f2476ffcd1f4fe540e5a8489322d398be3c  
 4e5a869db7fcb7c52c  
 Output = c748ca6dd327f0ce85f4ae3a8cd6d4d5390bbb804c9e12dcf94f853fece  
 3dcce

### A.3.2. VOPRF Mode

Seed = a3a  
 3a3  
 KeyInfo = 74657374206b6579  
 skSm = ca5d94c8807817669a51b196c34c1b7f8442fde4334a7121ae4736364312f  
 ca6  
 pkSm = 03e17e70604bcabe198882c0a1f27a92441e774224ed9c702e51dd17038b1  
 02462

### A.3.2.1. Test Vector 1, Batch Size 1

Input = 00  
 Blind = 3338fa65ec36e0290022b48eb562889d89dbfa691d1cde91517fa222ed7a  
 d364  
 BlindedElement = 02dd05901038bb31a6fae01828fd8d0e49e35a486b5c5d4b499  
 4013648c01277da  
 EvaluationElement = 0209f33cab60cf8fe69239b0afbcfcd261af4c1c5632624f  
 2e9ba29b90ae83e4a2  
 Proof = e7c2b3c5c954c035949f1f74e6bce2ed539a3be267d1481e9ddb178533df  
 4c2664f69d065c604a4fd953e100b856ad83804eb3845189babfa5a702090d6fc5fa  
 ProofRandomScalar = f9db001266677f62c095021db018cd8cbb55941d4073698c  
 e45c405d1348b7b1  
 Output = 0412e8f78b02c415ab3a288e228978376f99927767ff37c5718d420010a  
 645a1



**A.3.2.2. Test Vector 2, Batch Size 1**

Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a  
Blind = 3338fa65ec36e0290022b48eb562889d89dbfa691d1cde91517fa222ed7ad364  
BlindedElement = 03cd0f033e791c4d79dfa9c6ed750f2ac009ec46cd4195ca6fd3800d1e9b887dbd  
EvaluationElement = 030d2985865c693bf7af47ba4d3a3813176576383d19aff003ef7b0784a0d83cf1  
Proof = 2787d729c57e3d9512d3aa9e8708ad226bc48e0f1750b0767aaff73482c44b8d2873d74ec88aebd3504961acea16790a05c542d9fbff4fe269a77510db00abab  
ProofRandomScalar = f9db001266677f62c095021db018cd8cbb55941d4073698ce45c405d1348b7b1  
Output = 771e10dcd6bcd3664e23b8f2a710cfaaa8357747c4a8cbbba03133967b5c24f18

**A.3.2.3. Test Vector 3, Batch Size 2**

Input = 00,5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a  
Blind = 3338fa65ec36e0290022b48eb562889d89dbfa691d1cde91517fa222ed7ad364, f9db001266677f62c095021db018cd8cbb55941d4073698ce45c405d1348b7b1  
BlindedElement = 02dd05901038bb31a6fae01828fd8d0e49e35a486b5c5d4b4994013648c01277da, 03462e9ae64cae5b83ba98a6b360d942266389ac369b923eb3d557213b1922f8ab  
EvaluationElement = 0209f33cab60cf8fe69239b0afbcfcd261af4c1c5632624f2e9ba29b90ae83e4a2, 02bb24f4d838414aef052a8f044a6771230ca69c0a5677540fff738dd31bb69771  
Proof = bdcc351707d02a72ce49511c7db990566d29d6153ad6f8982fad2b435d6ce4d60da1e6b3fa740811bde34dd4fe0aa1b5fe6600d0440c9ddee95ea7fad7a60cf2  
ProofRandomScalar = 350e8040f828bf6ceca27405420cdf3d63cb3aef005f40ba51943c8026877963  
Output = 0412e8f78b02c415ab3a288e228978376f99927767ff37c5718d420010a645a1, 771e10dcd6bcd3664e23b8f2a710cfaaa8357747c4a8cbbba03133967b5c24f18

**A.3.3. POPRF Mode**

Seed = a3  
KeyInfo = 74657374206b6579  
skSm = 6ad2173efa689ef2c27772566ad7ff6e2d59b3b196f00219451fb2c89ee4dae2  
pkSm = 030d7ff077fddeec965db14b794f0cc1ba9019b04a2f4fcc1fa525dedf72e2a3e3



#### A.4. P384-SHA384

##### A.4.1. OPRF Mode

Seed = a3  
3a3  
KeyInfo = 74657374206b6579  
skSm = dfe7ddc41a4646901184f2b432616c8ba6d452f9bcd0c4f75a5150ef2b2ed  
02ef40b8b92f60ae591bcabd72a6518f188

##### A.4.1.1. Test Vector 1, Batch Size 1

Input = 00  
Blind = 504650f53df8f16f6861633388936ea23338fa65ec36e0290022b48eb562  
889d89dbfa691d1cde91517fa222ed7ad364  
BlindedElement = 02a36bc90e6db34096346eaf8b7bc40ee1113582155ad379700  
3ce614c835a874343701d3f2debdb80d97cbe45de6e5f1f  
EvaluationElement = 03af2a4fc94770d7a7bf3187ca9cc4faf3732049eded2442  
ee50fbddda58b70ae2999366f72498cdbc43e6f2fc184afe30  
Output = ed84ad3f31a552f0456e58935fcc0a3039db42e7f356dcb32aa6d487b6b  
815a07d5813641fb1398c03ddab5763874357

##### A.4.1.2. Test Vector 2, Batch Size 1

Input = 5a  
Blind = 504650f53df8f16f6861633388936ea23338fa65ec36e0290022b48eb562  
889d89dbfa691d1cde91517fa222ed7ad364  
BlindedElement = 02def6f418e3484f67a124a2ce1bfb19de7a4af568ede6a1ebb  
2733882510ddd43d05f2b1ab5187936a55e50a847a8b900  
EvaluationElement = 034e9b9a2960b536f2ef47d8608b21597ba400d5abfa1825  
fd21c36b75f927f396bf3716c96129d1fa4a77fa1d479c8d7b  
Output = dd4f29da869ab9355d60617b60da0991e22aaab243a3460601e48b07585  
9d1c526d36597326f1b985778f781a1682e75

##### A.4.2. VOPRF Mode

Seed = a3a  
3a3  
KeyInfo = 74657374206b6579  
skSm = 051646b9e6e7a71ae27c1e1d0b87b4381db6d3595eeeb1adb41579adbf992  
f4278f9016eafc944edaa2b43183581779d  
pkSm = 031d689686c611991b55f1a1d8f4305ccd6cb719446f660a30db61b7aa87b  
46acf59b7c0d4a9077b3da21c25dd482229a0

**A.4.2.1. Test Vector 1, Batch Size 1**

Input = 00  
Blind = 504650f53df8f16f6861633388936ea23338fa65ec36e0290022b48eb562  
889d89dbfa691d1cde91517fa222ed7ad364  
BlindedElement = 02d338c05cbeeb82de13d6700f09cb61190543a7b7e2c6cd4fc  
a56887e564ea82653b27fdad383995ea6d02cf26d0e24d9  
EvaluationElement = 02a7bba589b3e8672aa19e8fd258de2e6aae20101c8d7612  
46de97a6b5ee9cf105febce4327a326255a3c604f63f600ef6  
Proof = bfc6cf3859127f5fe25548859856d6b7fa1c7459f0ba5712a806fc091a30  
00c42d8ba34ff45f32a52e40533efd2a03bc87f3bf4f9f58028297ccb9ccb18ae718  
2bcd1ef239df77e3be65ef147f3acf8bc9cbfc5524b702263414f043e3b7ca2e  
ProofRandomScalar = 803d955f0e073a04aa5d92b3fb739f56f9db001266677f62  
c095021db018cd8cbb55941d4073698ce45c405d1348b7b1  
Output = 333230886b562ffb8329a8be08fea8025755372817ec969d114d1203d0  
26b4a622beab60220bf19078bca35a529b35c

**A.4.2.2. Test Vector 2, Batch Size 1**

Input = 5a  
Blind = 504650f53df8f16f6861633388936ea23338fa65ec36e0290022b48eb562  
889d89dbfa691d1cde91517fa222ed7ad364  
BlindedElement = 02f27469e059886f221be5f2cca03d2bdc61e55221721c3b3e5  
6fc012e36d31ae5f8dc058109591556a6dbd3a8c69c433b  
EvaluationElement = 03f16f903947035400e96b7f531a38d4a07ac89a80f89d86  
a1bf089c525a92c7f4733729ca30c56ce78b1ab4f7d92db8b4  
Proof = d005d6daaad7571414c1e0c75f7e57f2113ca9f4604e84bc90f9be52da89  
6fff3bee496dcde2a578ae9df315032585f801fb21c6080ac05672b291e575a40295  
b306d967717b28e08fcc8ad1cab47845d16af73b3e643ddcc191208e71c64630  
ProofRandomScalar = 803d955f0e073a04aa5d92b3fb739f56f9db001266677f62  
c095021db018cd8cbb55941d4073698ce45c405d1348b7b1  
Output = b91c70ea3d4d62ba922eb8a7d03809a441e1c3c7af915cbc2226f485213  
e895942cd0f8580e6d99f82221e66c40d274f

**A.4.2.3. Test Vector 3, Batch Size 2**

Input = 00,5a  
Blind = 504650f53df8f16f6861633388936ea23338fa65ec36e0290022b48eb562  
889d89dbfa691d1cde91517fa222ed7ad364, 803d955f0e073a04aa5d92b3fb739f5  
6f9db001266677f62c095021db018cd8cbb55941d4073698ce45c405d1348b7b1  
BlindedElement = 02d338c05cbeeb82de13d6700f09cb61190543a7b7e2c6cd4fc  
a56887e564ea82653b27fdad383995ea6d02cf26d0e24d9, 02fa02470d7f151018b4  
1e82223c32fad824de6ad4b5ce9f8e9f98083c9a726de9a1fc39d7a0cb6f4f188dd9  
cea01474cd  
EvaluationElement = 02a7bba589b3e8672aa19e8fd258de2e6aae20101c8d7612  
46de97a6b5ee9cf105febce4327a326255a3c604f63f600ef6, 028e9e115625ff4c2  
f07bf87ce3fd73fc77994a7a0c1df03d2a630a3d845930e2e63a165b114d98fe34e6  
1b68d23c0b50a  
Proof = 6d8dcbd2fc95550a02211fb78afd013933f307d21e7d855b0b1ed0af7807  
6d8137ad8b0a1bfa05676d325249c1dbb9a52bd81b1c2b7b0efc77cf7b278e1c947f  
6283f1d4c513053fc0ad19e026fb0c30654b53d9cea4b87b037271b5d2e2d0ea  
ProofRandomScalar = a097e722ed2427de86966910acba9f5c350e8040f828bf6c  
eca27405420cdf3d63cb3aef005f40ba51943c8026877963  
Output = 333230886b562ffb8329a8be08fea8025755372817ec969d114d1203d0  
26b4a622beab60220bf19078bca35a529b35c, b91c70ea3d4d62ba922eb8a7d03809  
a441e1c3c7af915cbc2226f485213e895942cd0f8580e6d99f82221e66c40d274f











### A.5.3. POPRF Mode

Seed = a3  
3a3  
KeyInfo = 74657374206b6579  
skSm = 014893130030ce69cf714f536498a02ff6b396888f9bb507985c32928c442  
7d6d39de10ef509aca4240e8569e3a88debc0d392e3361bcd934cb9bdd59e339dff7  
b27  
pkSm = 0301de8ceb9ffe9237b1bba87c320ea0bebcfc3447fe6f278065c6c69886d  
692d1126b79b6844f829940ace9b52a5e26882cf7cbc9e57503d4cca3cd834584729  
f812a

#### A.5.3.1. Test Vector 1, Batch Size 1

Input = 00  
Info = 7465737420696e666f  
Blind = 00d1dccf7a51bafaf75d4a866d53d8cafe4d504650f53df8f16f68616333  
88936ea23338fa65ec36e0290022b48eb562889d89dbfa691d1cde91517fa222ed7a  
d364  
BlindedElement = 020095cff9d7ecf65bdfef4ea92d6e748d60b02de34ad98094f  
82e25d33a8bf50138ccc2cc633556f1a97d7ea9438cbb394df612f041c485a515849  
d5ebb2238f2f0e2  
EvaluationElement = 0301408e9c5be3ffcc1c16e5ae8f8aa68446223b0804b119  
62e856af5a6d1c65ebbb5db7278c21db4e8cc06d89a35b6804fb1738a295b691638a  
f77aa1327253f26d01  
Proof = 0106a89a61eee9dd2417d2849a8e2167bc5f56e3aed5a3ff23e22511fa1b  
37a29ed44d1bbfd6907d99cfbc558a56aec709282415a864a281e49dc53792a4a638  
a0660034306d64be12a94dcea5a6d664cf76681911c8b9a84d49bf12d4893307ec14  
436bd05f791f82446c0de4be6c582d373627b51886f76c4788256e3da7ec8fa18a86  
ProofRandomScalar = 015e80ae32363b32cb76ad4b95a5a34e46bb803d955f0e07  
3a04aa5d92b3fb739f56f9db001266677f62c095021db018cd8cbb55941d4073698c  
e45c405d1348b7b1  
Output = 808ae5b87662eaaf0b39151dd85991b94c96ef214cb14a68bf5c1439548  
82d330da8953a80eea20788e552bc8bbbf3100e89f9d6e341197b122c46a208733  
b

### A.5.3.2. Test Vector 2, Batch Size 1

Input = 5a  
Info = 7465737420696e666f  
Blind = 00d1dccb7a51bafaf75d4a866d53d8cafe4d504650f53df8f16f68616333  
88936ea23338fa65ec36e0290022b48eb562889d89dbfa691d1cde91517fa222ed7a  
d364  
BlindedElement = 030112ea89cf9cf589496189eafc5f9eb13c9f9e170d6ecde7c  
5b940541cb1a9c5cfeec908b67efe16b81ca00d0ce216e34b3d5f46a658d3fd8573d  
671bdb6515ed508  
EvaluationElement = 0200ebc49df1e6fa61f412e6c391e6f074400ecdd2f56c4a  
8c03fe0f91d9b551f40d4b5258fd891952e8c9b28003bcfa365122e54a5714c8949d  
5d202767b31b4bf1f6  
Proof = 0082162c71a7765005cae202d4bd14b84dae63c29067e886b82506992bd9  
94a1c3aac0c1c5309222fe1af8287b6443ed6df5c2e0b0991faddd3564c73c7597ae  
cd9a003b1f1e3c65f28e58ab4e767cfb4adbcdf512441645f4c2aed8bf67d132d966  
006d35fa71a34145414bf3572c1de1a46c266a344dd9e22e7fb1e90ffba1caf556d9  
ProofRandomScalar = 015e80ae32363b32cb76ad4b95a5a34e46bb803d955f0e07  
3a04aa5d92b3fb739f56f9db001266677f62c095021db018cd8cbb55941d4073698c  
e45c405d1348b7b1  
Output = 27032e24b1a52a82ab7f4646f3c5df0f070f499db98b9c5df33972bd5af  
5762c3638afae7912a6c1acdb1ae2ab2fa670bd5486c645a0e55412e08d33a4a0d6e  
3

### A.5.3.3. Test Vector 3, Batch Size 2

Input = 00,5a  
Info = 7465737420696e666f  
Blind = 00d1dccb7a51bafaf75d4a866d53d8cafe4d504650f53df8f16f68616333  
88936ea23338fa65ec36e0290022b48eb562889d89dbfa691d1cde91517fa222ed7a  
d364,015e80ae32363b32cb76ad4b95a5a34e46bb803d955f0e073a04aa5d92b3fb7  
39f56f9db001266677f62c095021db018cd8cbb55941d4073698ce45c405d1348b7b  
1  
BlindedElement = 020095cff9d7ecf65bdfec4ea92d6e748d60b02de34ad98094f  
82e25d33a8bf50138ccc2cc633556f1a97d7ea9438cbb394df612f041c485a515849  
d5ebb2238f2f0e2,0201a328cf9f3fdeb86b6db242dd4cbb436b3a488b70b72d2fbb  
d1e5f50d7b0878b157d6f278c6a95c488f3ad52d6898a421658a82fe7ceb000b01ae  
dea7967522d525  
EvaluationElement = 0301408e9c5be3ffcc1c16e5ae8f8aa68446223b0804b119  
62e856af5a6d1c65ebbb5db7278c21db4e8cc06d89a35b6804fb1738a295b691638a  
f77aa1327253f26d01,020062ab51ac3aa829e0f5b7ae50688bcf5f63a18a83a6e0d  
a538666b8d50c7ea2b4ef31f4ac669302318dbebe46660acdda695da30c22cee7ca2  
1f6984a720504502e  
Proof = 00731738844f739bca0cca9d1c8bea204bed4fd00285785738b985763741  
de5cdfa275152d52b6a2fdf7792ef3779f39ba34581e56d62f78ecad5b7f8083f384  
961501cd4b43713253c022692669cf076b1d382ecd8293c1de69ea569737f37a2477  
2ab73517983c1e3db5818754ba1f008076267b8058b6481949ae346cdc17a8455fe2  
ProofRandomScalar = 01ec21c7bb69b0734cb48dfd68433dd93b0fa097e722ed24  
27de86966910acba9f5c350e8040f828bf6ceca27405420cdf3d63cb3aef005f40ba  
51943c8026877963  
Output = 808ae5b87662eaa0f0b39151dd85991b94c96ef214cb14a68bf5c1439548  
82d330da8953a80eea20788e552bc8bbbfff3100e89f9d6e341197b122c46a208733  
b,27032e24b1a52a82ab7f4646f3c5df0f070f499db98b9c5df33972bd5af5762c36  
38afae7912a6c1acdb1ae2ab2fa670bd5486c645a0e55412e08d33a4a0d6e3

## Authors' Addresses

Alex Davidson  
Brave Software

Email: [alex.davidson92@gmail.com](mailto:alex.davidson92@gmail.com)

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

Email: [armfazh@cloudflare.com](mailto:armfazh@cloudflare.com)

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

Email: [nick@cloudflare.com](mailto:nick@cloudflare.com)

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

Email: [caw@heapingbits.net](mailto:caw@heapingbits.net)