

Network Working Group  
Internet-Draft  
Intended status: Informational  
Expires: 12 August 2022

A. Davidson  
Brave Software  
A. Faz-Hernandez  
N. Sullivan  
C.A. Wood  
Cloudflare, Inc.  
8 February 2022

Oblivious Pseudorandom Functions (OPRFs) using Prime-Order Groups  
draft-irtf-cfrg-voprf-09

## Abstract

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

## Discussion Venues

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

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

## Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of [BCP 78](#) and [BCP 79](#).

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

Internet-Draft

OPRFs

February 2022

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 12 August 2022.

## Copyright Notice

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

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

## Table of Contents

<a href="#">1.</a>	Introduction . . . . .	<a href="#">4</a>
<a href="#">1.1.</a>	Change log . . . . .	<a href="#">4</a>
<a href="#">1.2.</a>	Requirements . . . . .	<a href="#">7</a>
<a href="#">1.3.</a>	Notation and Terminology . . . . .	<a href="#">7</a>
<a href="#">2.</a>	Preliminaries . . . . .	<a href="#">8</a>
<a href="#">2.1.</a>	Prime-Order Group . . . . .	<a href="#">8</a>
<a href="#">2.2.</a>	Discrete Log Equivalence Proofs . . . . .	<a href="#">10</a>
<a href="#">2.2.1.</a>	Proof Generation . . . . .	<a href="#">10</a>
<a href="#">2.2.2.</a>	Proof Verification . . . . .	<a href="#">13</a>
<a href="#">3.</a>	Protocol . . . . .	<a href="#">16</a>
<a href="#">3.1.</a>	Configuration . . . . .	<a href="#">18</a>
<a href="#">3.2.</a>	Key Generation and Context Setup . . . . .	<a href="#">18</a>
<a href="#">3.3.</a>	Online Protocol . . . . .	<a href="#">20</a>
<a href="#">3.3.1.</a>	OPRF Protocol . . . . .	<a href="#">21</a>
<a href="#">3.3.2.</a>	VOPRF Protocol . . . . .	<a href="#">23</a>
<a href="#">3.3.3.</a>	POPRF Protocol . . . . .	<a href="#">24</a>
<a href="#">4.</a>	Ciphersuites . . . . .	<a href="#">27</a>
<a href="#">4.1.</a>	Ciphersuite Registry . . . . .	<a href="#">28</a>
<a href="#">4.1.1.</a>	OPRF(ristretto255, SHA-512) . . . . .	<a href="#">28</a>
<a href="#">4.1.2.</a>	OPRF(dec448, SHAKE-256) . . . . .	<a href="#">29</a>

<a href="#">4.1.3.</a>	<a href="#">OPRF(P-256, SHA-256)</a>	<a href="#">29</a>
<a href="#">4.1.4.</a>	<a href="#">OPRF(P-384, SHA-384)</a>	<a href="#">30</a>
<a href="#">4.1.5.</a>	<a href="#">OPRF(P-521, SHA-512)</a>	<a href="#">30</a>
<a href="#">4.2.</a>	<a href="#">Input Validation</a>	<a href="#">31</a>
<a href="#">4.2.1.</a>	<a href="#">DeserializeElement Validation</a>	<a href="#">31</a>

<a href="#">4.2.2.</a>	<a href="#">DeserializeScalar Validation</a>	<a href="#">31</a>
<a href="#">4.3.</a>	<a href="#">Future Ciphersuites</a>	<a href="#">32</a>
<a href="#">5.</a>	<a href="#">Application Considerations</a>	<a href="#">32</a>
<a href="#">5.1.</a>	<a href="#">Input Limits</a>	<a href="#">32</a>
<a href="#">5.2.</a>	<a href="#">External Interface Recommendations</a>	<a href="#">32</a>
<a href="#">5.3.</a>	<a href="#">Error Considerations</a>	<a href="#">33</a>
<a href="#">5.4.</a>	<a href="#">POPRF Public Input</a>	<a href="#">33</a>
<a href="#">6.</a>	<a href="#">Security Considerations</a>	<a href="#">34</a>
<a href="#">6.1.</a>	<a href="#">Security Properties</a>	<a href="#">34</a>
<a href="#">6.2.</a>	<a href="#">Security Assumptions</a>	<a href="#">35</a>
<a href="#">6.2.1.</a>	<a href="#">OPRF and VOPRF Assumptions</a>	<a href="#">35</a>
<a href="#">6.2.2.</a>	<a href="#">POPRF Assumptions</a>	<a href="#">36</a>
<a href="#">6.2.3.</a>	<a href="#">Static Diffie Hellman Attack and Security Limits</a>	<a href="#">36</a>
<a href="#">6.3.</a>	<a href="#">Domain Separation</a>	<a href="#">37</a>
<a href="#">6.4.</a>	<a href="#">Timing Leaks</a>	<a href="#">37</a>
<a href="#">7.</a>	<a href="#">Acknowledgements</a>	<a href="#">37</a>
<a href="#">8.</a>	<a href="#">References</a>	<a href="#">37</a>
<a href="#">8.1.</a>	<a href="#">Normative References</a>	<a href="#">37</a>
<a href="#">8.2.</a>	<a href="#">Informative References</a>	<a href="#">38</a>
<a href="#">Appendix A.</a>	<a href="#">Test Vectors</a>	<a href="#">40</a>
<a href="#">A.1.</a>	<a href="#">OPRF(ristretto255, SHA-512)</a>	<a href="#">41</a>
<a href="#">A.1.1.</a>	<a href="#">OPRF Mode</a>	<a href="#">41</a>
<a href="#">A.1.2.</a>	<a href="#">VOPRF Mode</a>	<a href="#">41</a>
<a href="#">A.1.3.</a>	<a href="#">POPRF Mode</a>	<a href="#">43</a>
<a href="#">A.2.</a>	<a href="#">OPRF(dec448, SHAKE-256)</a>	<a href="#">44</a>
<a href="#">A.2.1.</a>	<a href="#">OPRF Mode</a>	<a href="#">44</a>
<a href="#">A.2.2.</a>	<a href="#">VOPRF Mode</a>	<a href="#">45</a>
<a href="#">A.2.3.</a>	<a href="#">POPRF Mode</a>	<a href="#">47</a>
<a href="#">A.3.</a>	<a href="#">OPRF(P-256, SHA-256)</a>	<a href="#">49</a>
<a href="#">A.3.1.</a>	<a href="#">OPRF Mode</a>	<a href="#">49</a>
<a href="#">A.3.2.</a>	<a href="#">VOPRF Mode</a>	<a href="#">50</a>
<a href="#">A.3.3.</a>	<a href="#">POPRF Mode</a>	<a href="#">51</a>
<a href="#">A.4.</a>	<a href="#">OPRF(P-384, SHA-384)</a>	<a href="#">53</a>
<a href="#">A.4.1.</a>	<a href="#">OPRF Mode</a>	<a href="#">53</a>
<a href="#">A.4.2.</a>	<a href="#">VOPRF Mode</a>	<a href="#">54</a>
<a href="#">A.4.3.</a>	<a href="#">POPRF Mode</a>	<a href="#">55</a>

<a href="#">A.5.</a>	OPRF(P-521, SHA-512)	<a href="#">57</a>
<a href="#">A.5.1.</a>	OPRF Mode	<a href="#">57</a>
<a href="#">A.5.2.</a>	VOPRF Mode	<a href="#">58</a>
<a href="#">A.5.3.</a>	POPRF Mode	<a href="#">60</a>
Authors' Addresses		<a href="#">62</a>

## [1.](#) Introduction

A Pseudorandom Function (PRF)  $F(k, x)$  is an efficiently computable function taking a private key  $k$  and a value  $x$  as input. This function is pseudorandom if the keyed function  $K(\_) = F(k, \_)$  is indistinguishable from a randomly sampled function acting on the same domain and range as  $K()$ . An Oblivious PRF (OPRF) is a two-party protocol between a server and a client, where the server holds a PRF key  $k$  and the client holds some input  $x$ . The protocol allows both parties to cooperate in computing  $F(k, x)$  such that the client learns  $F(k, x)$  without learning anything about  $k$ ; and the server does not learn anything about  $x$  or  $F(k, x)$ . A Verifiable OPRF (VOPRF) is an OPRF wherein the server also proves to the client that  $F(k, x)$  was produced by the key  $k$  corresponding to the server's public key the client knows. A Partially-Oblivious PRF (POPRF) is a variant of a VOPRF wherein client and server interact in computing  $F(k, x, y)$ , for some PRF  $F$  with server-provided key  $k$ , client-provided input  $x$ , and public input  $y$ , and client receives proof that  $F(k, x, y)$  was computed using  $k$  corresponding to the public key that the client knows. A POPRF with fixed input  $y$  is functionally equivalent to a VOPRF.

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

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

### 1.1. Change log

draft-09 (<https://tools.ietf.org/html/draft-irtf-cfrg-voprf-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.

Davidson, et al.

Expires 12 August 2022

[Page 4]

---

Internet-Draft

OPRFs

February 2022

- \* Simplify security considerations.
- \* Update application interface considerations.
- \* Update test vectors.

draft-08 (<https://tools.ietf.org/html/draft-irtf-cfrg-voprf-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 (<https://tools.ietf.org/html/draft-irtf-cfrg-voprf-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 (<https://tools.ietf.org/html/draft-irtf-cfrg-voprf-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 (<https://tools.ietf.org/html/draft-irtf-cfrg-voprf-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 (<https://tools.ietf.org/html/draft-irtf-cfrg-voprf-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 (<https://tools.ietf.org/html/draft-irtf-cfrg-voprf-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](https://tools.ietf.org/html/draft-irtf-cfrg-voprf-02) (<https://tools.ietf.org/html/draft-irtf-cfrg-voprf-02>):

- \* Added section discussing cryptographic security and static DH oracles.
- \* Updated batched proof algorithms.

[draft-01](https://tools.ietf.org/html/draft-irtf-cfrg-voprf-01) (<https://tools.ietf.org/html/draft-irtf-cfrg-voprf-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.
- \* I2OSP and OS2IP: Convert a byte array to and from a non-negative integer as described in [[RFC8017](#)]. Note that these functions operate on byte arrays in big-endian byte order.



in a Python-like pseudocode. The data types `PrivateInput` and `PublicInput` are opaque byte strings of arbitrary length no larger than  $2^{13}$  octets.

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

The following terms are used throughout this document.

- \* PRF: Pseudorandom Function.
- \* OPRF: Oblivious Pseudorandom Function.
- \* VOPRF: Verifiable Oblivious Pseudorandom Function.
- \* POPRF: Partially Oblivious Pseudorandom Function.
- \* Client: Protocol initiator. Learns pseudorandom function evaluation as the output of the protocol.
- \* Server: Computes the pseudorandom function over a private key. Learns nothing about the client's input or output.

## [2.](#) Preliminaries

The protocols in this document have two primary dependencies:

- \* Group: A prime-order group implementing the API described below in [Section 2.1](#), with base point defined in the corresponding reference for each group. (See [Section 4](#) for these base points.)
- \* Hash: A cryptographic hash function whose output length is  $N_h$  bytes long.

[Section 4](#) specifies ciphersuites as combinations of Group and Hash.

### [2.1.](#) Prime-Order Group

In this document, we assume the construction of an additive, prime-order group `Group` for performing all mathematical operations. Such groups are uniquely determined by the choice of the prime  $p$  that defines the order of the group. (There may, however, exist different representations of the group for a single  $p$ . [Section 4](#) lists specific groups which indicate both order and representation.) We use  $\text{GF}(p)$  to represent the finite field of order  $p$ . For the purpose of understanding and implementing this document, we take  $\text{GF}(p)$  to be equal to the set of integers defined by  $\{0, 1, \dots, p-1\}$ .

The fundamental group operation is addition  $+$  with identity element  $I$ . For any elements  $A$  and  $B$  of the group,  $A + B = B + A$  is also a member of the group. Also, for any  $A$  in the group, there exists an element  $-A$  such that  $A + (-A) = (-A) + A = I$ . Scalar multiplication is equivalent to the repeated application of the group operation on an element  $A$  with itself  $r-1$  times, this is denoted as  $r*A = A + \dots + A$ . For any element  $A$ ,  $p*A=I$ . Scalar base multiplication is equivalent to the repeated application of the group operation on the fixed group generator with itself  $r-1$  times, and is denoted as  $\text{ScalarBaseMult}(r)$ . The set of scalars corresponds to  $\text{GF}(p)$ . This document uses types `Element` and `Scalar` to denote elements of the group and its set of scalars, respectively.

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

- \* `Order()`: Outputs the order of the group (i.e.  $p$ ).
- \* `Identity()`: Outputs the identity element of the group (i.e.  $I$ ).
- \* `HashToGroup(x)`: A member function of `Group` that deterministically maps an array of bytes  $x$  to an element of `Group`. The map must ensure that, for any adversary receiving  $R = \text{HashToGroup}(x)$ , it is computationally difficult to reverse the mapping. This function is optionally parameterized by a domain separation tag (DST); see [Section 4](#).
- \* `HashToScalar(x)`: A member function of `Group` that 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](#).
- \* `RandomScalar()`: A member function of `Group` that chooses at random a non-zero element in  $\text{GF}(p)$ .
- \* `ScalarInverse(s)`: Compute the multiplicative inverse of input `Scalar s` modulo the prime order of the group  $p$ .
- \* `SerializeElement(A)`: A member function of `Group` that maps a group element  $A$  to a unique byte array `buf` of fixed length  $N_e$ . The output type of this function is `SerializedElement`.
- \* `DeserializeElement(buf)`: A member function of `Group` that maps a byte array `buf` to a group element  $A$ , or fails if the input is not a valid byte representation of an element. This function can raise a `DeserializeError` if deserialization fails or  $A$  is the identity element of the group; see [Section 4.2](#).

- \* `SerializeScalar(s)`: A member function of `Group` that maps a scalar element `s` to a unique byte array `buf` of fixed length `Ns`. The output type of this function is `SerializedScalar`.
- \* `DeserializeScalar(buf)`: A member function of `Group` that maps a byte array `buf` to a scalar `s`, or fails if the input is not a valid byte representation of a scalar. This function can raise a `DeserializeError` if deserialization fails; see [Section 4.2](#).

It is convenient in cryptographic applications to instantiate such prime-order groups using elliptic curves, such as those detailed in [\[SEC2\]](#). For some choices of elliptic curves (e.g. those detailed in [\[RFC7748\]](#), which require accounting for cofactors) there are some implementation issues that introduce inherent discrepancies between standard prime-order groups and the elliptic curve instantiation. In this document, all algorithms that we detail assume that the group is a prime-order group, and this MUST be upheld by any implementation. That is, any curve instantiation should be written such that any discrepancies with a prime-order group instantiation are removed. See [Section 4](#) for advice corresponding to the implementation of this interface for specific definitions of elliptic curves.

## [2.2](#). Discrete Log Equivalence Proofs

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

### [2.2.1](#). Proof Generation

Generating a proof is done with the `GenerateProof` function, defined below. This function takes four Elements, `A`, `B`, `C`, and `D`, and a single group Scalar `k`, and produces a proof that  $k * A == B$  and  $k * C ==$

D. The output is a value of type Proof, which is a tuple of two Scalar values.

Input:

Scalar k  
Element A  
Element B  
Element C  
Element D

Output:

Proof proof

Parameters:

Group G

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

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

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

    h2Input = I2OSP(len(Bm), 2) || Bm ||
```

```

        I2OSP(len(a0), 2) || a0 ||
        I2OSP(len(a1), 2) || a1 ||
        I2OSP(len(a2), 2) || a2 ||
        I2OSP(len(a3), 2) || a3 ||
        "Challenge"

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

    return [c, s]

```

The helper function `ComputeCompositesFast` is as defined below.

#### Input:

```

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

```

#### Output:

```

    Element M
    Element Z

```

#### Parameters:

```

    Group G
    PublicInput contextString

```

```

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

    M = G.Identity()

```

```

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

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

Z = k * M

return (M, Z)

```

When used in the protocol described in [Section 3](#), the parameter contextString is as defined in [Section 3.1](#).

ComputeCompositesFast takes lists of inputs, rather than a single input. Applications can take advantage of this functionality by invoking GenerateProof on batches of inputs to produce a combined, constant-size proof. In particular, servers can produce a single, constant-sized proof for N DLEQ inputs, rather than one proof per

DLEQ input. This optimization benefits clients and servers since it amortizes the cost of proof generation and bandwidth across multiple requests.

### [2.2.2](#). Proof Verification

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

Internet-Draft

OPRFs

February 2022

Input:

Element A  
Element B  
Element C  
Element D  
Proof proof

Output:

boolean verified

Parameters:

Group G

Errors: DeserializeError

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

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

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

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

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

    expectedC = G.HashToScalar(h2Input)

    return expectedC == c
```

The definition of ComputeComposites is given below.

Input:

Element B

Element Cs[m]



Element Ds[m]

Output:

Element M  
Element Z

Parameters:

Group G  
PublicInput contextString

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

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

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

    return (M, Z)
```

When used in the protocol described in [Section 3](#), the parameter contextString is as defined in [Section 3.1](#).

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

### 3. Protocol

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

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

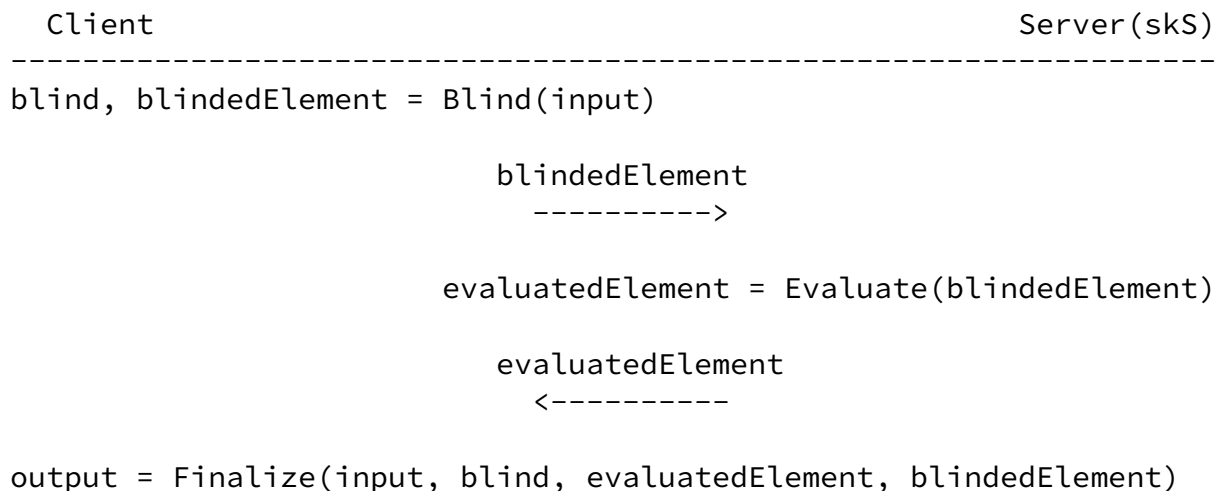


Figure 1: OPRF protocol overview

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

Internet-Draft

OPRFs

February 2022

```

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

                        blindedElement
                        ----->

evaluatedElement, proof = Evaluate(blindedElement)

                        evaluatedElement, proof
                        <-----

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

```

Figure 2: VOPRF protocol overview with additional proof

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

```

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

                        blindedElement
                        ----->

evaluatedElement, proof = Evaluate(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:

+=====+=====+	
Mode	Value
+=====+=====+	
modeOPRF	0x00
+-----+-----+	
modeVOPRF	0x01
+-----+-----+	
modePOPRF	0x02
+-----+-----+	

Table 1

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

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

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

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

### [3.2](#). Key Generation and Context Setup

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

Input:

opaque seed[Ns]  
PublicKey info

Output:

Scalar skS  
Element pkS

Parameters:

Group G  
PublicKey contextString

Errors: DeriveKeyPairError

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

```

        DST = "DeriveKeyPair" || contextString)
    counter = counter + 1
    pkS = G.ScalarBaseMult(skS)
    return skS, pkS

```

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

```

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

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

```

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

```

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

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

```

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

```

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

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

```

### [3.3.](#) Online Protocol

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

- \*  $G$ , a prime-order Group implementing the API described in [Section 2.1](#).
- \* `contextString`, a `PublicInput` domain separation tag constructed during context setup as created in [Section 3.1](#).
- \*  $sk_S$  and  $pk_S$ , a Scalar and Element representing the private and public keys configured for client and server in [Section 3.2](#).

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

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

### [3.3.1](#). OPRF Protocol

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

Input:

`PrivateInput` input

Output:

Scalar blind  
Element blindedElement

Parameters:

Group G

Errors: InvalidInputError

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

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

Input:

Element blindedElement

Output:



Element evaluatedElement

Parameters:

Scalar skS

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

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

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

Input:

PrivateInput input  
Scalar blind  
Element evaluatedElement

Output:

opaque output[Nh]

Parameters:

Group G

```
def Finalize(input, blind, evaluatedElement):  
    N = G.ScalarInverse(blind) * evaluatedElement  
    unblindedElement = G.SerializeElement(N)  
  
    hashInput = I2OSP(len(input), 2) || input ||  
                I2OSP(len(unblindedElement), 2) || unblindedElement ||  
                "Finalize"  
    return Hash(hashInput)
```

### [3.3.2.](#) VOPRF Protocol

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

Input:

Element blindedElement

Output:

Element evaluatedElement  
Proof proof

Parameters:

Group G  
Scalar skS  
Element pkS

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

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

Internet-Draft

OPRFs

February 2022

Input:

```
PrivateInput input
Scalar blind
Element evaluatedElement
Element blindedElement
Proof proof
```

Output:

```
opaque output[Nh]
```

Parameters:

```
Group G
Element pkS
```

Errors: VerifyError

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

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

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

### [3.3.3.](#) POPRF Protocol

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

---

Internet-Draft

OPRFs

February 2022

Input:

```
PrivateInput input
PublicInput info
```

Output:

```
Scalar blind
Element blindedElement
Element tweakedKey
```

Parameters:

```
Group G
Element pkS
```

Errors: InvalidInputError

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

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

    blindedElement = blind * P

    return blind, blindedElement, tweakedKey
```

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

Input:

Element `blindedElement`  
PublicInput `info`

Output:

Element `evaluatedElement`  
Proof `proof`

Parameters:

Group `G`  
Scalar `skS`  
Element `pkS`

Errors: `InverseError`

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

    evaluatedElement = G.ScalarInverse(t) * blindedElement

    tweakedKey = G.ScalarBaseMult(t)
```

```

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

return evaluatedElement, proof

```

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

#### Input:

```

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

```

#### Output:

```

opaque output[Nh]

```

#### Parameters:

```

Group G
Element pkS

```

#### Errors: `VerifyError`

```

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

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

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

```

#### 4. Ciphersuites

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

A ciphersuite contains instantiations of the following functionalities:

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

This section specifies an initial registry of ciphersuites with

supported groups and hash functions. It also includes implementation details for each ciphersuite, focusing on input validation, as well as requirements for future ciphersuites.

#### [4.1.](#) Ciphersuite Registry

For each ciphersuite, contextString is that which is computed in the Setup functions. Applications should take caution in using ciphersuites targeting P-256 and ristretto255. See [Section 6.2](#) for related discussion.

##### [4.1.1.](#) OPRF(ristretto255, SHA-512)

- \* Group: ristretto255 [[RISTRETTO](#)]
  - HashToGroup(): Use hash\_to\_ristretto255 [[I-D.irtf-cfrg-hash-to-curve](#)] with DST = "HashToGroup-" || contextString, and expand\_message = expand\_message\_xmd using SHA-512.
  - HashToScalar(): Compute uniform\_bytes using expand\_message = expand\_message\_xmd, DST = "HashToScalar-" || contextString, and output length 64, interpret uniform\_bytes as a 512-bit integer in little-endian order, and reduce the integer modulo Order().
  - Serialization: Both group elements and scalars are encoded in Ne = Ns = 32 bytes. For group elements, use the 'Encode' and 'Decode' functions from [[RISTRETTO](#)]. For scalars, ensure they are fully reduced modulo Order() and in little-endian order.
- \* Hash: SHA-512, and Nh = 64.
- \* ID: 0x0001

##### [4.1.2.](#) OPRF(decaf448, SHAKE-256)

- \* Group: decaf448 [[RISTRETTO](#)]
  - HashToGroup(): Use hash\_to\_decaf448 [[I-D.irtf-cfrg-hash-to-curve](#)] with DST = "HashToGroup-" || contextString, and expand\_message = expand\_message\_xof using



SHAKE-256.

- HashToScalar(): Compute uniform\_bytes using expand\_message = expand\_message\_xof, DST = "HashToScalar-" || contextString, and output length 64, interpret uniform\_bytes as a 512-bit integer in little-endian order, and reduce the integer modulo Order().
  - Serialization: Both group elements and scalars are encoded in Ne = Ns = 56 bytes. For group elements, use the 'Encode' and 'Decode' functions from [\[RISTRETTO\]](#). For scalars, ensure they are fully reduced modulo Order() and in little-endian order.
- \* Hash: SHAKE-256, and Nh = 64.
- \* ID: 0x0002

#### [4.1.3.](#) OPRF(P-256, SHA-256)

- \* Group: P-256 (secp256r1) [\[x9.62\]](#)
- HashToGroup(): Use hash\_to\_curve with suite P256\_XMD:SHA-256\_SSWU\_RO\_ [\[I-D.irtf-cfrg-hash-to-curve\]](#) and DST = "HashToGroup-" || contextString.
  - HashToScalar(): Use hash\_to\_field from [\[I-D.irtf-cfrg-hash-to-curve\]](#) using L = 48, expand\_message\_xmd with SHA-256, DST = "HashToScalar-" || contextString, and prime modulus equal to Order().
  - Serialization: Elements are serialized as Ne = 33 byte strings using compressed point encoding for the curve [\[SEC1\]](#). Scalars are serialized as Ns = 32 byte strings by fully reducing the value modulo Order() and in big-endian order.
- \* Hash: SHA-256, and Nh = 32.
- \* ID: 0x0003

#### [4.1.4.](#) OPRF(P-384, SHA-384)

- \* Group: P-384 (secp384r1) [[x9.62](#)]
  - HashToGroup(): Use hash\_to\_curve with suite P384\_XMD:SHA-384\_SSWU\_RO\_ [[I-D.irtf-cfrg-hash-to-curve](#)] and DST = "HashToGroup-" || contextString.
  - HashToScalar(): Use hash\_to\_field from [[I-D.irtf-cfrg-hash-to-curve](#)] using L = 72, expand\_message\_xmd with SHA-384, DST = "HashToScalar-" || contextString, and prime modulus equal to Order().
  - Serialization: Elements are serialized as Ne = 49 byte strings using compressed point encoding for the curve [[SEC1](#)]. Scalars are serialized as Ns = 48 byte strings by fully reducing the value modulo Order() and in big-endian order.
- \* Hash: SHA-384, and Nh = 48.
- \* ID: 0x0004

#### [4.1.5.](#) OPRF(P-521, SHA-512)

- \* Group: P-521 (secp521r1) [[x9.62](#)]
  - HashToGroup(): Use hash\_to\_curve with suite P521\_XMD:SHA-512\_SSWU\_RO\_ [[I-D.irtf-cfrg-hash-to-curve](#)] and DST = "HashToGroup-" || contextString.
  - HashToScalar(): Use hash\_to\_field from [[I-D.irtf-cfrg-hash-to-curve](#)] using L = 98, expand\_message\_xmd with SHA-512, DST = "HashToScalar-" || contextString, and prime modulus equal to Order().
  - Serialization: Elements are serialized as Ne = 67 byte strings using compressed point encoding for the curve [[SEC1](#)]. Scalars are serialized as Ns = 66 byte strings by fully reducing the value modulo Order() and in big-endian order.
- \* Hash: SHA-512, and Nh = 64.
- \* ID: 0x0005

Internet-Draft

OPRFs

February 2022

## [4.2.](#) Input Validation

The `DeserializeElement` and `DeserializeScalar` functions instantiated for a particular prime-order group corresponding to a ciphersuite MUST adhere to the description in [Section 2.1](#). This section describes how both `DeserializeElement` and `DeserializeScalar` are implemented for all prime-order groups included in the above ciphersuite list.

### [4.2.1.](#) `DeserializeElement` Validation

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

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

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

### [4.2.2.](#) `DeserializeScalar` Validation

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

For P-256, P-384, and P-521 ciphersuites, this function ensures that the input, when treated as a big-endian integer, is a value between 0 and `Order() - 1`. For `ristretto255` and `decaf448`, this function ensures that the input, when treated as a little-endian integer, is a value between 0 and `Order() - 1`.

### [4.3.](#) Future Ciphersuites

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

In the security proof of the construction `Hash` is modeled as a random oracle. This implies that any instantiation of `HashToGroup` must be pre-image and collision resistant. In [Section 4](#) we give instantiations of this functionality based on the functions described in [\[I-D.irtf-cfrg-hash-to-curve\]](#). Consequently, any OPRF implementation must adhere to the implementation and security considerations discussed in [\[I-D.irtf-cfrg-hash-to-curve\]](#) when instantiating the function.

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

## [5.](#) Application Considerations

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

### [5.1.](#) Input Limits

Application inputs, expressed as `PrivateInput` or `PublicInput` values, MUST be smaller than  $2^{13}$  bytes in length. Applications that require longer inputs can use a cryptographic hash function to map these longer inputs to a fixed-length input that fits within the `PublicInput` or `PrivateInput` length bounds. Note that some cryptographic hash functions have input length restrictions themselves, but these limits are often large enough to not be a

concern in practice. For example, SHA-256 has an input limit of  $2^{61}$  bytes.

## [5.2.](#) External Interface Recommendations

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

## [5.3.](#) Error Considerations

Some OPRF variants specified in this document have fallible operations. For example, Finalize and Evaluate can fail if any element received from the peer fails deserialization. The explicit errors generated throughout this specification, along with the conditions that lead to each error, are as follows:

- \* `VerifyError`: Verifiable OPRF proof verification failed; [Section 3.3.2](#) and [Section 3.3.3](#).
- \* `DeserializeError`: Group Element or Scalar deserialization failure; [Section 2.1](#) and [Section 3.3](#).

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

- \* `InvalidInputError`: OPRF Blind input produces an invalid output element; [Section 3.3.1](#) and [Section 3.3.3](#).
- \* `InverseError`: A tweaked private key is invalid (has no multiplicative inverse); [Section 2.1](#) and [Section 3.3](#).

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

## 5.4. POPRF Public Input

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

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

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

## 6. Security Considerations

This section discusses the cryptographic security of our protocol, along with some suggestions and trade-offs that arise from the implementation of the OPRF variants in this document. Note that the syntax of the POPRF variant is different from that of the OPRF and POPRF variants since it admits an additional public input, but the same security considerations apply.

### 6.1. Security Properties

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

- \* Pseudorandomness:  $F$  is pseudorandom if the output  $y = F(k, x)$  on any input  $x$  is indistinguishable from uniformly sampling any

element in  $F$ 's range, for a random sampling of  $k$ .

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

A consequence of showing that a function is pseudorandom, is that it is necessarily non-malleable (i.e. we cannot compute a new evaluation of  $F$  from an existing evaluation). A genuinely random function will be non-malleable with high probability, and so a pseudorandom function must be non-malleable to maintain indistinguishability.

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

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

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

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

Any VOPRF or POPRF that satisfies the 'verifiable' security property is known as 'verifiable'. In practice, the notion of verifiability requires that the server commits to the key before the actual protocol execution takes place. Then the client verifies that the server has used the key in the protocol using this commitment. In the following, we may also refer to this commitment as a public key.

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

- \* Partial obliviousness: The server must learn nothing about the

client's private input or the output of the function. In addition, the client must learn nothing about the server's private key. Both client and server learn the public input (info).

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

## [6.2.](#) Security Assumptions

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

### [6.2.1.](#) OPRF and VOPRF Assumptions

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

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

### [6.2.2.](#) POPRF Assumptions

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



Pseudorandomness, input secrecy, verifiability, and partial obliviousness of the POPRF variant is based on the assumption that the One-More Gap Strong Diffie-Hellman Inversion (SDHI) assumption from [TCRSTW21] is computationally difficult to solve in the corresponding prime-order group. [TCRSTW21] show that both the One-More Gap CDH assumption and the One-More Gap SDHI assumption reduce to the q-DL (Discrete Log) assumption in the algebraic group model, for some q number of Evaluate queries. (The One-More Gap CDH assumption was the hardness assumption used to evaluate the OPRF and VOPRF designs based on [JKK14], which is a predecessor to the POPRF variant in [Section 3.3.3](#).)

### [6.2.3](#). Static Diffie Hellman Attack and Security Limits

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

As a result of this class of attack, choosing prime-order groups with a 128-bit security level instantiates an OPRF with a reduced security level of  $128 - (\log_2(Q)/2)$  bits of security. Moreover, such attacks are only possible for those certain applications where the adversary can query the OPRF directly. Applications can mitigate against this problem in a variety of ways, e.g., by rate-limiting client queries to Evaluate() or by rotating private keys. In applications where such an oracle is not made available this security loss does not apply.

In most cases, it would require an informed and persistent attacker to launch a highly expensive attack to reduce security to anything much below 100 bits of security. Applications that admit the aforementioned oracle functionality, and that cannot tolerate discrete logarithm security of lower than 128 bits, are RECOMMENDED to choose groups that target a higher security level, such as decaf448 (used by ciphersuite 0x0002), P-384 (used by 0x0004), or P-521 (used by 0x0005).

## [6.3](#). Domain Separation

Applications SHOULD construct input to the protocol to provide domain separation. Any system which has multiple OPRF applications should distinguish client inputs to ensure the OPRF results are separate. Guidance for constructing info can be found in [\[I-D.irtf-cfrg-hash-to-curve\]](#), Section 3.1.

#### [6.4.](#) Timing Leaks

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

### [7.](#) Acknowledgements

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

### [8.](#) References

#### [8.1.](#) Normative References

[I-D.ietf-privacypass-protocol]

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

[I-D.irtf-cfrg-hash-to-curve]

Faz-Hernandez, A., Scott, S., Sullivan, N., Wahby, R. S., and C. A. Wood, "Hashing to Elliptic Curves", Work in Progress, Internet-Draft, [draft-irtf-cfrg-hash-to-curve-13](#), 10 November 2021, <<https://www.ietf.org/archive/id/draft-irtf-cfrg-hash-to-curve-13.txt>>.

[I-D.irtf-cfrg-opaque]

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

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

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

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

[RISTRETTO]

Valence, H. D., Grigg, J., Tankersley, G., Valsorda, F., Lovecruft, I., and M. Hamburg, "The ristretto255 and decaf448 Groups", Work in Progress, Internet-Draft, [draft-irtf-cfrg-ristretto255-decaf448-01](https://www.ietf.org/archive/id/draft-irtf-cfrg-ristretto255-decaf448-01), 4 August 2021, <<https://www.ietf.org/archive/id/draft-irtf-cfrg-ristretto255-decaf448-01.txt>>.

## [8.2](#). Informative References

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

[ChaumPedersen]

"Wallet Databases with Observers", n.d., <[https://chaum.com/publications/Wallet\\_Databases.pdf](https://chaum.com/publications/Wallet_Databases.pdf)>.

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

[DGSTV18] "Privacy Pass, Bypassing Internet Challenges Anonymously", <<https://www.degruyter.com/view/j/popets.2018.2018.issue-3/popets-2018-0026/popets-2018-0026.xml>>.

Internet-Draft

OPRFs

February 2022

- [JKK14] "Round-Optimal Password-Protected Secret Sharing and T-PAKE in the Password-Only model", <<https://eprint.iacr.org/2014/650>>.
- [JKKX16] "Highly-Efficient and Composable Password-Protected Secret Sharing (Or, How to Protect Your Bitcoin Wallet Online)", <<https://eprint.iacr.org/2016/144>>.
- [keyagreement]  
Barker, E., Chen, L., Roginsky, A., Vassilev, A., and R. Davis, "Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography", DOI 10.6028/nist.sp.800-56ar3, National Institute of Standards and Technology report, April 2018, <<https://doi.org/10.6028/nist.sp.800-56ar3>>.
- [PrivacyPass]  
"Privacy Pass", <<https://github.com/privacypass/challenge-bypass-server>>.
- [RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", [RFC 7748](#), DOI 10.17487/RFC7748, January 2016, <<https://www.rfc-editor.org/info/rfc7748>>.
- [SEC1] Standards for Efficient Cryptography Group (SECG), ., "SEC 1: Elliptic Curve Cryptography", <<https://www.secg.org/sec1-v2.pdf>>.
- [SEC2] Standards for Efficient Cryptography Group (SECG), ., "SEC 2: Recommended Elliptic Curve Domain Parameters", <<http://www.secg.org/sec2-v2.pdf>>.
- [SJKS17] "SPHINX, A Password Store that Perfectly Hides from Itself", <<https://eprint.iacr.org/2018/695>>.
- [TCRSTW21] "A Fast and Simple Partially Oblivious PRF, with Applications", <<https://eprint.iacr.org/2021/864>>.
- [x9.62] ANSI, "Public Key Cryptography for the Financial Services

## [Appendix A](#). Test Vectors

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

- \* "Input": The private client input, an opaque byte string.
- \* "Info": The public info, an opaque byte string. Only present for POPRF vectors.
- \* "Blind": The blind value output by Blind(), a serialized Scalar of  $N_s$  bytes long.
- \* "BlindedElement": The blinded value output by Blind(), a serialized Element of  $N_e$  bytes long.
- \* "EvaluatedElement": The evaluated element output by Evaluate(), a serialized Element of  $N_e$  bytes long.
- \* "Proof": The serialized Proof output from GenerateProof() (only listed for verifiable mode test vectors), composed of two serialized Scalar values each of  $N_s$  bytes long. Only present for VOPRF and POPRF vectors.
- \* "ProofRandomScalar": The random scalar  $r$  computed in GenerateProof() (only listed for verifiable mode test vectors), a serialized Scalar of  $N_s$  bytes long. Only present for VOPRF and

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

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

The server key material,  $pk_{Sm}$  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](#).

Internet-Draft OPRFs February 2022

### A.1.1. OPRF Mode

#### A.1.1.1. Test Vector 1, Batch Size 1

#### A.1.1.2. Test Vector 2, Batch Size 1











[illegible]

#### A.2.1.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = d62851d4bc07947c858dc735e9e22aaf0576f161ab555182908dbd7947b1
c988956fa73b17b373b72fd4e3c0264a26aa4cab20fd6193b933
BlindedElement = d078a185d2d8a54b68d6df4e83640192d3659e18fec68d43e48
02998d3c9fd819b32070caa78083c909d68daeb7fd420a73f931452a2b70d
EvaluationElement = 3452e46b6277b032627a7e5d22aa1b25459f8de90dda3137
9ed490bb0078eeec05fc4265fafbb5252d4228f9f1f5453bbd391d6b8589f232
Output = b93d3ed18489c1236cc965d202254de35767ea673560d6c225cec0b30fe
3adc88fee63f8a78d127cd64c7077e1d3ac4a7cc761335c0bcddc12d6981ad8730285
8
```

#### A.2.1.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = ac345e8d755997956ddd1f267a2d86175aeae5e1168932285a6f602b4b20
a570a697452b3ddb7d0e29363adebbcb5673294396b82931f37
BlindedElement = 283f0fab2be6ac3a3c8eacfd504f3ef63f518892f7b000f1dcc
1ca2e773aba0fbee48b100886b90d5a08377cbf5ccf69801ae2c23e1adbf2
EvaluationElement = ae30bab51a34c45a76d00034b29e1c5346fbe3718c386302
8e47226456880a85a2e5118f274a8c260dae62fcec3cde8624405fc7cddbc867
Output = aaf99e5a044bbce915bf3ba381e25da62e4b2cea4cee2f47f3662940284
579c0f8e1e011062ba010ca4f2c67a8157481c9ae7a458ea035a89e1948bfc5b8323
b
```

### A.2.2. VOPRF Mode

[illegible]

#### A.2.2.1. Test Vector 1, Batch Size 1

Internet-Draft

OPRFs

February 2022

```
Input = 00
Blind = 4bdfc97a75132d92a1da241baff84fada3e7b12d5b712efcac9ba734d54c
2b24bff0ef6310404b5c05d60d7c258cea6500229ee015149f0f
BlindedElement = 1ceb0a3432ac6b583c31fa70b7c17ac86e0aa425e0593d04b58
021670f725eee6664e6cd2041d90f157bc213a2aa4ed7929630b2d9898a76
EvaluationElement = 3afaa02425294a4810766c68e9e4c3c507b109b9064ed56a
148a419371d5fb158f6ab5f0da62a6ba915bbe431097f5c71854821c1f10889d
Proof = f02f7ab2722508e343b5692078556e7ca9b2d63bf83dff902150b867775b
f375693cc6a0adf33178ba7e72d6179b36ed051065c93619752958746f0d52e2e3a9
89d86df15f458847abdcc23976147b7b10c96452332aa03bfce1b89b7aead080869d
7ce8c7acb7414e7dbfcda298b532
ProofRandomScalar = 54534ad9db9f6df6ce515d1b8017923b65cada199e936a62
3c8eb3bd08e9b3f6584a85e4ff26e9f869d30b6c7c6cc56fd94e306974fbcc3b
Output = b558e37f6435a12fefded196936a4c1d0882bf4a115002920744ecb3128
43678f396f7d36711cf551750388ddf7a53a3aea7fd0ac60568cd2d4ead16a1ee106
f
```

#### [A.2.2.2.](#) Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = beda1edc786e5fd0033feac1992c53a607d516a46251614940e76a2763b8
0683e5b789398710bdbbc774d9221dd33c509b4805fc26f0c8d0b
BlindedElement = 2e04b6883057a5b5ba020d077ae36dee76a07c2f3eb8cc55baf
dfb3da9c7405ffe50802f646ca3c3ef39d195c2d88ee56e73825c7cd2319a
EvaluationElement = 6270b2f73738aa846dca34d7b30b7c4f943e31d4d4fb35c5
98f5d608cf25648b44553d43b158dc2707eda170dc439740c10d7b4355bf0f83
Proof = f731f60aa18d508f07dd3b7851fe9f8cfe6f02c4ea2814cfe8af3203e493
44041e6acf0f09fdffdc02d22728544b9bda8d0604e727f27a1efa16526f169191de
db35a1338bf399d8737d6d1638f6d4b895c0869b4194e66fb0dbb4b3e0437a2af0d7
6dd8cfb0bf38c9de605dc5749603
ProofRandomScalar = 00cc800042a0cff31f865698f8858efa75a1f0faef934317
dd6a10bfbbbb39f9f2d97dcd5ff4eae02980b08fc68da7b71d39399dc4eb0400a
Output = eb14608be2f14c25b2c9fdd23690d293d0c6aaac501a3405b626b8699cf
34bb9dd4c2d7987b6391519b9480da453611509ba98098b3e79a35acd00f5e9d8abc
e
```

#### [A.2.2.3.](#) Test Vector 3, Batch Size 2

February 2022

### A.2.3. POPRF Mode

[illegible]

#### [A.2.3.1.](#) Test Vector 1, Batch Size 1

Davidson, et al.

Expires 12 August 2022

[Page 47]

---

Internet-Draft

OPRFs

February 2022

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

#### [A.2.3.2.](#) Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Info = 7465737420696e666f
Blind = 1abe4937f28f531b14ac96b844320e7a66810c2d9391cbb877348301ab59
a3a91b4a2129672886ae5da7839f2ac8cf1c5fa92703f5b3fd06
BlindedElement = dea615b00285247715173fc6db40cab1436607bc0eaed3d7a1a
1467b70c7ff2f2ce91c05bcaeda2b01952926f254f13e1a763a174caa693a
```



[illegible]



### A.3.2. VOPRF Mode

[illegible]

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

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

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

[illegible]



Input = 00  
Info = 7465737420696e666f  
Blind = 4238835743037876080d2e3e27bc3ce7b5fb6a1107ffedeaedb371767432b68c  
BlindedElement = 02cb57f07ba100b93ce1bf8176963c8c7f73a76827f1c1401a923d7ca4083e15aa  
EvaluationElement = 03059d58ec9a801e33f57525c03241d8ffb61b67a18edd35222d864ffbb42b5d2f  
Proof = 13889d6849850ccd0119981fc053a38a30a57d275091df2887943d1332f738204f8a6cf2fb6e57c9b118ec82b9b012f8864561e4cd8866245f9c762b9d45dbf9  
ProofRandomScalar = 3d5c65b55a1b8960563b3420d7764097502850c445ccd86e2d20d7e4ec77617b  
Output = 15fce9922a2307349aac2eccc41941283e3c5e938aaf2506f99a6d8b6ee34ef8

#### [A.3.3.2.](#) Test Vector 2, Batch Size 1

Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a  
Info = 7465737420696e666f  
Blind = c262bf51dc970d63acb5ab74318e54223c759e9747f59c0d4ecbc087302667fb  
BlindedElement = 02208809631cc08f553d7843db566c55746e760a77c63513d2b22096f98452cf9d  
EvaluationElement = 02301ee1cf1d01276649ac0f718ebbfaf1c0d6a1b3e7ea82b3085e9173910fcb0ef  
Proof = cb69a1ec76643a2100cc9bfe6cf1ed1fa5ba3612ed3e3211036b5ed835a138be3eb92126694e3e925ab138d4df885be18ed80371847f80baab82ce70588eebaf  
ProofRandomScalar = 6c6990f0fcd9a655f77ff0b2ebcfe21e1a1ca4a84361e9f1b18e24c9a40ed5ef  
Output = a06ed7380210856caaba173bcad06266186c6638d86e372c3c96b9bd2f353543

#### [A.3.3.3.](#) Test Vector 3, Batch Size 2

February 2022

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

Internet-Draft

## OPRFs

February 2022

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = f9e066cf04a050c4fd762bfff10c1b9bd5d37afc6f3644f8545b9a09a6d7a
3073b3c9b3d78588213957ea3a5dfd0f1fe4
BlindedElement = 023c36bf6352c93d27b118972d1040cf22f99d5a1c8134afb89
8d30b319f70a096973db23410881f84eea599c0c73220bd
EvaluationElement = 0240b6a002d0190793ea62a7499244027753d63b0a57cea1
98c8c6dc883cdeb273ab385699bb414f1040bb6819313cd675
Output = 1d155a7ba2ea75c4f1e76fb0a37231e9b0776eed3f24a6541a01907ca8a
fb984a74408e6d2de8e481cae5dd03bdae3ce
```

#### A.4.2. VOPRF Mode

[illegible]

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

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



```
Proof = 1e26bf1210717b88dfae585008100e9ccaeb93b8605ca168a608cbf1855
697b7b87d0b9c6bdca85e43143b3630e87f2fe9ce519dba3d477d2a869bcad0db9dc
6239cd11938213f9bfd63d39de090a6fc90cd1f33f164b2c54c38bc31ad98ddf
ProofRandomScalar = b36f4c2a140b7a3c53dd8efb6171d3bb4d73591be8483a1a
38e40c13a04b0f2180dda3c36e3d43c3a8f127158d010945
Output = f18884ace2e342f849cea7f2f17de902b9884574fdaa8f507356f482c6b
67013f329e8c899b3c2c154af1defaa11d656,f91d172cdecdea4f8299c8b39426db
4c47428b82f8872b8539ad9b019deb48b8d3c928c572ed988d5591a4442c060438
```

[illegible]

[Page 55]

February 2022

```
Input = 00
Info = 7465737420696e666f
Blind = 9572d3a8a106f875023c9722b2de94efaa02c8e46a9e48f3e2ee00241f9a
75f3f7493200a8a605644334de4987fb60da
BlindedElement = 0252f98f04a956afa469c62ca2850f751b112dc019d4e713c66
2fc0735ef8573f1497cea55b750f27f0efc8330e394a3ab
EvaluationElement = 03fb20c33a7f6f01f2bb388318a6db84f7183bc3bd5e5840
302fe38b6b313649b523238b4c4c625614440dd6ddbcbcc7272
Proof = d33c83c1840a48759659a4d417769ae3bb1adb86326a36fa1ff24f70066b
75d0200e5c1e7d9847e91f7d3d6843efc62101c401a7c952cde32ada6fec848450d8
564e2c778af47ece4f50a88c6d2281bdd858b90fdfad8b093c986bc1e59aaa2e
ProofRandomScalar = 7e82569cb56d97e9c20e59311bac3a50735d573abb787b25
1879b77de4df554c91e25e117919a9db2af19b32ce0d501d
Output = af52cf184180177970be0770e1c7920aa307b767556a13de38a64723d8d
cc7b344af9b6dd8f117ac2cef249ee3acc8fb
```

[illegible]











February 2022

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

Internet-Draft

## OPRFs

February 2022

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

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

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Info = 7465737420696e666f
Blind = 0085ad3fc8c91caec3bd7699591b10d6da93877a470e128f38030627dffcbbf1f576b38677841fc47af778f9d85ac9bce6279388ddf4607e295e64cea6f4f95078b8
BlindedElement = 0201eeaedeb3692cc0ecfeacdf9cab61947eb0d23bbfe2e1fbf8de0907f9410b6089d060d3af63411fd81b9d588fa2c48bf8ec63ec66c14b86d237124042ca83fc99e1
EvaluationElement = 0300886138e19945036ebe6f4195cf9f688d9e5a7c89597dfee6a6e0e5fcf4b53a9dfa280c8409b6abe8051e3394279d0b669440af8a27aad169de10446eb88e09d6801
Proof = 01cb4d8a14eeb472ee3e2fbfe3f6d49f3654cfe6238254bea17ce30848ca
```

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

[illegible]

## Authors' Addresses

Alex Davidson  
Brave Software

Email: alex.davidson92@gmail.com

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

Email: armfazh@cloudflare.com

Davidson, et al.

Expires 12 August 2022

[Page 62]

---

Internet-Draft

OPRFs

February 2022

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

Email: nick@cloudflare.com

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

Email: caw@heapingbits.net

