

Internet Research Task Force (IRTF)
Request for Comments: 9497
Category: Informational
ISSN: 2070-1721

A. Davidson
Brave Software
A. Faz-Hernandez
N. Sullivan
C. A. Wood
Cloudflare, Inc.
December 2023

Oblivious Pseudorandom Functions (OPRFs) Using Prime-Order Groups

Abstract

An Oblivious Pseudorandom Function (OPRF) is a two-party protocol between a client and a server for computing the output of a Pseudorandom Function (PRF). The server provides the PRF private 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 private 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.

Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This document is a product of the Internet Research Task Force (IRTF). The IRTF publishes the results of Internet-related research and development activities. These results might not be suitable for deployment. This RFC represents the consensus of the Crypto Forum Research Group of the Internet Research Task Force (IRTF). Documents approved for publication by the IRSG are not candidates for any level of Internet Standard; see Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at <https://www.rfc-editor.org/info/rfc9497>.

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.

Table of Contents

1. Introduction
 - 1.1. Requirements Language

1.2.	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.	References	
8.1.	Normative References	
8.2.	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	
	Acknowledgements	
	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, wherein 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, which the client knows. A Partially Oblivious PRF (POPRF) is a variant of a VOPRF, where the 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 the 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 (PAKE) [OPAQUE]. Verifiable OPRFs are necessary in some applications, such as Privacy Pass [PRIVACY-PASS]. 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. Requirements Language

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.

1.2. 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 nonnegative integer x into a byte array of specified length xLen , as described in [RFC8017]. Note that this function returns a byte array in big-endian byte order.
- * The notation $T\ U[N]$ refers to an array called U , containing N items of type T . The type opaque means one single byte of uninterpreted data. Items of the array are zero-indexed and referred to 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 the 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 PRF evaluation as the output of the protocol.

Server: Computes the PRF 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, denoted Group, 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. (However, different representations of the group for a single p may exist. Section 4 lists specific groups that indicate both the 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 is 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 [RFC9380].

`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 [RFC9380], Section 10.5.

`RandomScalar()`: Chooses at random a nonzero 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 Scalar s to a canonical byte array buf of fixed-length N_s .

`DeserializeScalar(buf)`: Attempts to map a byte array buf to 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 Pseudorandom Number Generator (PRNG). The description is split into two subsections: 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, as defined below. Given `Element` values `A` and `B`, two non-empty lists of `Element` values `C` and `D` of length `m`, and `Scalar` `k`, this function produces a proof that $k * A == B$ and $k * C[i] == D[i]$ for each `i` in `[0, ..., m - 1]`. The output is a value of type `Proof`, which is a tuple of two `Scalar` values. We use the notation `proof[0]` and `proof[1]` to denote the first and second elements in this tuple, respectively.

`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)

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

    M = G.Identity()
    for i in range(m):
        Ci = G.SerializeElement(C[i])
        Di = G.SerializeElement(D[i])
        compositeTranscript =
            I2OSP(len(seed), 2) || seed || I2OSP(i, 2) ||
            I2OSP(len(Ci), 2) || Ci ||
            I2OSP(len(Di), 2) || Di ||
            "Composite"

        di = G.HashToScalar(compositeTranscript)
        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, as defined below. This function takes Element values A and B, two non-empty lists of Element values 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)

    challengeTranscript =
        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(challengeTranscript)
    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
    seedTranscript =
        I2OSP(len(Bm), 2) || Bm ||
```



```

    I2OSP(len(seedDST), 2) || seedDST
seed = Hash(seedTranscript)

M = G.Identity()
Z = G.Identity()
for i in range(m):
    Ci = G.SerializeElement(C[i])
    Di = G.SerializeElement(D[i])
    compositeTranscript =
        I2OSP(len(seed), 2) || seed || I2OSP(i, 2) ||
        I2OSP(len(Ci), 2) || Ci ||
        I2OSP(len(Di), 2) || Di ||
        "Composite"

    di = G.HashToScalar(compositeTranscript)
    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 and describe three protocol variants referred to as the OPRF, VOPRF, and POPRF modes. Each of these variants involves two messages between the client and server, but they differ slightly in terms of the security properties; see Section 7.1 for more information. A high-level description of the functionality of each mode follows.

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 the 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 is 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 the public input info, which is used in computing the PRF. That is, the client and server interact to compute $\text{output} = F(\text{skS}, \text{input}, \text{info})$, 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, as described in Sections 3.2 and 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, as 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 32 bytes generated from a cryptographically secure random number generator and an optional (possibly empty) info string. 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[32]
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 the client and server in Section 3.2

Applications serialize protocol messages between the client and server for transmission. `Element` values and `Scalar` values are serialized to byte arrays, and values of type `Proof` are serialized as the concatenation of two serialized `Scalar` values. 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 that knows both the private 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 a 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 that knows both the private 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 a 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 private key of the server. Thus, clients that cause this error should be assumed to know the server private key. Hence, this error can be a signal for the server to replace its private 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 that knows both the private 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 [RFC9380], 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 includes an initial set of ciphersuites with supported groups and hash functions. It also includes implementation details for each ciphersuite, focusing on input validation. Future documents can specify additional ciphersuites as needed, provided they meet the requirements in Section 4.6.

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 [RFC9496] for the Group and SHA-512 for the hash function. The value of the ciphersuite identifier is "ristretto255-SHA512".

Group: ristretto255 [RFC9496]

Order(): Return $2^{252} + 27742317777372353535851937790883648493$ (see [RFC9496]).

Identity(): As defined in [RFC9496].

Generator(): As defined in [RFC9496].

HashToGroup(): Use hash_to_ristretto255 [RFC9380] 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 an output length of 64 bytes, 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 $s \bmod \text{Group.Order}()$.

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

SerializeElement(A): Implemented using the Encode function from Section 4.3.2 of [RFC9496]; $N_e = 32$.

DeserializeElement(buf): Implemented using the Decode function from Section 4.3.1 of [RFC9496]. 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; $N_s = 32$.

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 $[0, \text{G.Order}() - 1]$. 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 [RFC9496] for the Group and SHAKE-256 for the hash function. The value of the ciphersuite identifier is "decaf448-SHAKE256".

Group: decaf448 [RFC9496]

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

Identity(): As defined in [RFC9496].

Generator(): As defined in [RFC9496].

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

HashToGroup(): Use hash_to_decaf448 [RFC9380] 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 $s \bmod \text{Group.Order()}$.

SerializeElement(A): Implemented using the Encode function from Section 5.3.2 of [RFC9496]; Ne = 56.

DeserializeElement(buf): Implemented using the Decode function from Section 5.3.1 of [RFC9496]. 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 $[0, \text{G.Order()} - 1]$.

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 0xffffffff00000000ffffffffffffffffbce6faada7179e84f3b9cac2fc632551.

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_` [RFC9380] and `DST = "HashToGroup-" || contextString`.

HashToScalar(): Use `hash_to_field` from [RFC9380] using `L = 48`, `expand_message_xmd` with `SHA-256`, `DST = "HashToScalar-" || contextString`, and a 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 = 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 performing 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]; `Ns = 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; `Nh = 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 `0xfffc7634d81f4372ddf581a0db248b0a77aececl96accc52973`.

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_` [RFC9380] and `DST = "HashToGroup-" || contextString`.

HashToScalar(): Use `hash_to_field` from [RFC9380] using `L = 72`,

expand_message_xmd with SHA-384, DST = "HashToScalar-" || contextString, and a 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 = 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 performing 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 = 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; Nh = 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 0x01fffa51868783bf2f966b7fcc0148f709a5d03bb5c9b8899c47aebb6fb71e91386409.

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_ [RFC9380] and DST = "HashToGroup-" || contextString.

HashToScalar(): Use hash_to_field from [RFC9380] using L = 98, expand_message_xmd with SHA-512, DST = "HashToScalar-" || contextString, and a 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 67-byte input string to a public key using the compressed Octet-String-to-Elliptic-Curve-Point method according to [SEC1] and then performing 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]; $N_s = 66$.

`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; $N_h = 64$.

4.6. Future Ciphersuites

A critical requirement of implementing the prime-order group using elliptic curves is a method to instantiate the function `HashToGroup`, which 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 [RFC9380]. Consequently, any OPRF implementation must adhere to the implementation and security considerations discussed in [RFC9380] 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 described in the following subsections.

4.7.1. Rejection Sampling

Generate a random byte array with N_s 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 $L = \text{ceil}(((3 * \text{ceil}(\log_2(G.\text{Order()})) / 2) / 8)$ bytes, and interpret it as an integer; reduce the integer modulo $G.\text{Order}()$, and return the result. See [RFC9380], 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 `Element` and `Scalar` values. However, implementations can instead operate over `Element` and `Scalar` values internally and only expose interfaces that operate with an application-specific format of messages.

5.3. Error Considerations

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

`VerifyError`: Verifiable OPRF proof verification failed (Sections 3.3.2 and 3.3.3).

`DeserializeError`: Group `Element` or `Scalar` deserialization failure (Sections 2.1 and 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`: OPRF Blind input produces an invalid output element (Sections 3.3.1 and 3.3.3).

`InverseError`: A tweaked private key is invalid, i.e., has no multiplicative inverse (Sections 2.1 and 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 [RFC9380], 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 nonmalleable (i.e., we cannot compute a new evaluation of F from an existing evaluation). A genuinely random function will be nonmalleable with high probability, so a pseudorandom function must be nonmalleable 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 secrecy, 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 the client's private and public input. Both the 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 the 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, 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.
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 Diffie-Hellman (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 decaf448-SHAKE256), P-384 (used by ciphersuite P384-SHA384), or P-521 (used by ciphersuite P521-SHA512).

7.3. Domain Separation

Applications SHOULD construct input to the protocol to provide domain separation. Any system that has multiple OPRF applications should distinguish client inputs to ensure the OPRF results are separate. Guidance for constructing info can be found in [RFC9380], 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. References

8.1. Normative References

[KEYAGREEMENT]

Barker, E., Chen, L., Roginsky, A., Vassilev, A., and R. Davis, "Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography", NIST SP 800-56A (Rev. 3), 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/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>>.

[RFC9380] Faz-Hernandez, A., Scott, S., Sullivan, N., Wahby, R. S., and C. A. Wood, "Hashing to Elliptic Curves", RFC 9380, DOI 10.17487/RFC9380, August 2023, <<https://www.rfc-editor.org/info/rfc9380>>.

[RFC9496] de Valence, H., Grigg, J., Hamburg, M., Lovecruft, I., Tankersley, G., and F. Valsorda, "The ristretto255 and decaf448 Groups", RFC 9496, DOI 10.17487/RFC9496, December 2023, <<https://www.rfc-editor.org/info/rfc9496>>.

8.2. Informative References

[BG04] Brown, D. and R. Gallant, "The Static Diffie-Hellman Problem", November 2004, <<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 1992, <https://doi.org/10.1007/3-540-48071-4_7>.

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

- [DGSTV18] Davidson, A., Goldberg, I., Sullivan, N., Tankersley, G., and F. Valsorda, "Privacy Pass: Bypassing Internet Challenges Anonymously", Proceedings on Privacy Enhancing Technologies, vol. 2018, no. 3, pp. 164-180, DOI 10.1515/popets-2018-0026, April 2018, <<https://doi.org/10.1515/popets-2018-0026>>.
- [FS00] Fiat, A. and A. Shamir, "How To Prove Yourself: Practical Solutions to Identification and Signature Problems", Advances in Cryptology - CRYPTO' 86, pp. 186-194, DOI 10.1007/3-540-47721-7_12, 1986, <https://doi.org/10.1007/3-540-47721-7_12>.
- [JKK14] Jarecki, S., Kiayias, A., and H. Krawczyk, "Round-Optimal Password-Protected Secret Sharing and T-PAKE in the Password-Only Model", Lecture Notes in Computer Science, pp. 233-253, DOI 10.1007/978-3-662-45608-8_13, 2014, <https://doi.org/10.1007/978-3-662-45608-8_13>.
- [JKKX16] Jarecki, S., Kiayias, A., Krawczyk, H., and J. Xu, "Highly-Efficient and Composable Password-Protected Secret Sharing (Or: How to Protect Your Bitcoin Wallet Online)", 2016 IEEE European Symposium on Security and Privacy (EuroS&P), DOI 10.1109/eurosp.2016.30, March 2016, <<https://doi.org/10.1109/eurosp.2016.30>>.
- [NISTCurves] National Institute of Standards and Technology (NIST), "Digital Signature Standard (DSS)", FIPS PUB 186-5, DOI 10.6028/NIST.FIPS.186-5, February 2023, <<https://doi.org/10.6028/NIST.FIPS.186-5>>.
- [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-13, 18 December 2023, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-opaque-13>>.
- [PRIVACY-PASS] Celi, S., Davidson, A., Valdez, S., and C. A. Wood, "Privacy Pass Issuance Protocol", Work in Progress, Internet-Draft, draft-ietf-privacypass-protocol-16, 3 October 2023, <<https://datatracker.ietf.org/doc/html/draft-ietf-privacypass-protocol-16>>.
- [PrivacyPass] "Privacy Pass", commit 085380a, March 2018, <<https://github.com/privacypass/team>>.
- [RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <<https://www.rfc-editor.org/info/rfc7748>>.
- [SEC1] Standards for Efficient Cryptography Group (SECG), "SEC 1: Elliptic Curve Cryptography", May 2009, <<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", 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. A. 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, May 2022,
<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 running the OPRF, VOPRF, and POPRF modes. Each test vector lists the batch size for the evaluation. Each test vector value is encoded as a hexadecimal byte string. The fields of each test vector are described below.

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

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

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

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

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

"Proof": The serialized Proof output from GenerateProof() composed of two serialized Scalar values, each Ns 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 `Nh` bytes long.

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

The server key material, `pkSm` and `skSm`, are listed under the mode for each ciphersuite. Both `pkSm` and `skSm` are the serialized values of `pkS` and `skS`, respectively, as used in the protocol. Each key pair is derived from a seed, denoted `Seed`, and info string, denoted `KeyInfo`, which are listed as well, using the `DeriveKeyPair` function from Section 3.2.

A.1. ristretto255-SHA512

A.1.1. OPRF Mode

```
Seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3  
KeyInfo = 74657374206b6579  
skSm = 5ebcea5ee37023ccb9fc2d2019f9d7737be85591ae8652ffa9ef0f4d37063  
b0e
```

A.1.1.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = 64d37aed22a27f5191de1c1d69fadb899d8862b58eb4220029e036ec4c1f
```



```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a  
Info = 7465737420696e666f  
Blind = 64d37aed22a27f5191de1c1d69fad8b99d8862b58eb4220029e036ec4c1f6706  
BlindedElement = f0f0b209dd4d5f1844dac679acc7761b91a2e704879656cb7c201e82a99ab07d  
EvaluationElement = 8c3c9d064c334c6991e99f286ea2301dlbde170b54003fb9c44c6d7bd6fc1540  
Proof = 4c39992d55ffba38232cdac88fe583af8a85441fefed7dld4a8d0394cdlde77018bf135c174f20281b334lab1f453fe72b0293a7398703384bed822bfddec8908  
ProofRandomScalar = 222a5e897cf59db8145db8d16e597e8facb80ae7d4e26d9881aa6f61d645fc0e  
Output = 7c6557b276a137922a0bcfc2aa2b35dd78322bd500235eb6d6b6f91bc5b56a52de2d65612d503236b321f5d0bebbcb52b64b92e426f29c9b8b69f52de98ae507
```


A.1.3.3. Test Vector 3, Batch Size 2

[illegible]

A.2. decaf448-SHAKE256

A.2.1. OPRF Mode

[illegible]

A.2.1.1. Test Vector 1, Batch Size 1

```

Input = 00
Blind = 64d37aed22a27f5191de1c1d69fadb899d8862b58eb4220029e036ec65fa
3833a26e9388336361686ff1f83df55046504dfecad8549ba112
BlindedElement = e0ae01c4095f08e03b19baf47ffdc19cb7d98e583160522a3c7
d6a0b2111cd93a126a46b7b41b730cd7fc943d4e28e590ed33ae475885f6c
EvaluationElement = 50ce4e60eed006e22e7027454b5a4b8319eb2bc8ced609eb
19eb3ad42fb19e06ba12d382cbe7ae342a0cad6ead0ef8f91f00bb7f0cd9c0a2
Output = 37d3f7922d9388a15b561de5829bbf654c4089ede89c0ce0f3f85bcdba0
9e382ce0ab3507e021f9e79706a1798ffeac68ebd5cf62e5eb9838c7068351d97ae3
7

```

A.2.1.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a  
Blind = 64d37aed22a27f5191de1cld69fadfb899d8862b58eb4220029e036ec65fa  
3833a26e9388336361686ff1f83df55046504dfecad8549ba112  
BlindedElement = 86a88dc5c6331ecfcblD9aacb50a68213803c462e377577cacc  
00af28e15f0ddbc2e3d716f2f39ef95f3ec1314a2c64d940a9f295d8f13bb  
EvaluationElement = 162e9fa6e9d527c3cd734a31bf122a34dbd5bcb7bb23651f  
1768a7a9274cc116c03b58afa6f0dede3994a60066c76370e7328e7062fd5819  
Output = a2a652290055cb0f6f8637a249ee45e32ef4667db0b4c80c0a70d2a6416  
4d01525cfdad5d870a694ec77972b9b6ec5d2596a5223e5336913f945101f0137f55  
e
```

A.2.2. VOPRF Mode

[illegible]

A.2.3. POPRF Mode

[illegible]

A.2.3.1. Test Vector 1, Batch Size 1

```
Input = 00
Info = 7465737420696e666f
Blind = 64d37aed22a27f5191de1c1d69fadb899d8862b58eb4220029e036ec65fa
3833a26e9388336361686ff1f83df55046504dfecad8549ba112
BlindedElement = 161183c13c6cb33b0e4f9b7365f8c5c12d13c72f8b62d276ca0
9368d093dce9b42198276b9e9d870ac392dda53efd28d1b7e6e8c060cdc42
EvaluationElement = 06ec89dfde25bb2a6f0145ac84b91ac277b35de39ad1d6f4
02a8e46414952ce0d9ea1311a4ece283e2b01558c7078b040cfaa40dd63b3e6c
Proof = 66caee75bf2460429f620f6ad3e811d524cb8ddd848a435fc5d89af48877
abf6506ee341a0b6f67c2d76cd021e5f3d1c9abe5aa9f0dce016da746135fedba2af
41ed1d01659bfd6180d96bcb1b7f320c0cb6926011ce392ecca748662564892bae665
16acaac6ca39aadf6fcc9a5af406
ProofRandomScalar = b1b748135d405ce48c6973401d9455bb8ccd18b01d0295c0
627f67661200dbf9569f73fbb3925daa043a070e5f953d80bb464ea369e5522b
Output = 4423f6dcc1740688ea201de57d76824d59cd6b859e1f9884b7eebc49b0b
971358cf9cb075df1536a8ea31bcf55c3e31c2ba9cfa8efe54448d17091daeb9924e
d
```

A.2.3.2. Test Vector 2, Batch Size 1

[illegible]

A.2.3.3. Test Vector 3, Batch Size 2

[illegible]


```
ProofRandomScalar = 015e80ae32363b32cb76ad4b95a5a34e46bb803d955f0e07
3a04aa5d92b3fb739f56f9db001266677f62c095021db018cd8cbb55941d4073698c
e45c405d1348b7b1
Output = 27032e24b1a52a82ab7f4646f3c5df0f070f499db98b9c5df33972bd5af
5762c3638afae7912a6c1acdb1ae2ab2fa670bd5486c645a0e55412e08d33a4a0d6e
3
```

A.5.3.3. Test Vector 3, Batch Size 2

```
Input = 00,5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Info = 7465737420696e666f
Blind = 00d1dccb7a51bafaf75d4a866d53d8cafe4d504650f53df8f16f68616333
88936ea23338fa65ec36e0290022b48eb562889d89dbfa691d1cde91517fa222ed7a
d364,015e80ae32363b32cb76ad4b95a5a34e46bb803d955f0e073a04aa5d92b3fb7
39f56f9db001266677f62c095021db018cd8cbb55941d4073698ce45c405d1348b7b
1
BlindedElement = 020095cff9d7ecf65bdfee4ea92d6e748d60b02de34ad98094f
82e25d33a8bf50138ccc2cc633556f1a97d7ea9438cbb394df612f041c485a515849
d5ebb2238f2f0e2,0201a328cf9f3fdeb86b6db242dd4cbb436b3a488b70b72d2fbb
d1e5f50d7b0878b157d6f278c6a95c488f3ad52d6898a421658a82fe7ceb000b01ae
dea7967522d525
EvaluationElement = 0301408e9c5be3ffcc1c16e5ae8f8aa68446223b0804b119
62e856af5a6d1c65ebbb5db7278c21db4e8cc06d89a35b6804fb1738a295b691638a
f77aa1327253f26d01,020062ab51ac3aa829e0f5b7ae50688bcf5f63a18a83a6e0d
a538666b8d50c7ea2b4ef31f4ac669302318dbebe46660acd4695da30c22cee7ca2
1f6984a720504502e
Proof = 00731738844f739bca0cca9d1c8bea204bed4fd00285785738b985763741
de5cdfa275152d52b6a2fdf7792ef3779f39ba34581e56d62f78ecad5b7f8083f384
961501cd4b43713253c022692669cf076b1d382ecd8293c1de69ea569737f37a2477
2ab73517983c1e3db5818754ba1f008076267b8058b6481949ae346cdc17a8455fe2
ProofRandomScalar = 01ec21c7bb69b0734cb48dfd68433dd93b0fa097e722ed24
27de86966910acba9f5c350e8040f828bf6ceca27405420cdf3d63cb3aef005f40ba
51943c8026877963
Output = 808ae5b87662eaf0b39151dd85991b94c96ef214cb14a68bf5c1439548
82d330da89534a80eea20788e552bc8bbbf3100e89f9d6e341197b122c46a208733
b,27032e24b1a528ab7f4646f3c5df0f070f499db98b9c5df33972bd5af5762c36
38afae7912a6c1acd8b1ae2ab2fa670bd5486c645a0e55412e08d33a4a0d6e3
```

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.

Authors' Addresses

Alex Davidson
Brave Software
Email: alex.davidson92@gmail.com

Armando Faz-Hernandez
Cloudflare, Inc.
101 Townsend St
San Francisco, CA
United States of America
Email: armfazh@cloudflare.com

Nick Sullivan
Cloudflare, Inc.
101 Townsend St

San Francisco, CA
United States of America
Email: nicholas.sullivan+ietf@gmail.com

Christopher A. Wood
Cloudflare, Inc.
101 Townsend St
San Francisco, CA
United States of America
Email: caw@heapingbits.net