

Workgroup: CFRG  
Internet-Draft: draft-irtf-cfrg-vrf-15  
Published: 9 August 2022  
Intended Status: Informational  
Expires: 10 February 2023

A S. Goldberg L. Reyzin  
uBoston University Boston University and Algorand  
t  
h  
o  
r  
s  
:  
D. Papadopoulos  
Hong Kong University of Science and Technology  
J. Vcelak  
NS1

## Verifiable Random Functions (VRFs)

### Abstract

A Verifiable Random Function (VRF) is the public-key version of a keyed cryptographic hash. Only the holder of the secret key can compute the hash, but anyone with the public key can verify the correctness of the hash. VRFs are useful for preventing enumeration of hash-based data structures. This document specifies VRF constructions based on RSA and elliptic curves that are secure in the cryptographic random oracle model.

This document is a product of the Crypto Forum Research Group (CFRG) in the IRTF.

### Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

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

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

This Internet-Draft will expire on 10 February 2023.

### Copyright Notice

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

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents

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

## Table of Contents

- [1. Introduction](#)
  - [1.1. Requirements](#)
  - [1.2. Terminology](#)
- [2. VRF Algorithms](#)
- [3. VRF Security Properties](#)
  - [3.1. Full Uniqueness](#)
  - [3.2. Full Collision Resistance](#)
  - [3.3. Trusted Uniqueness and Trusted Collision Resistance](#)
  - [3.4. Full Pseudorandomness or Selective Pseudorandomness](#)
  - [3.5. Unpredictability Under Malicious Key Generation](#)
- [4. RSA Full Domain Hash VRF \(RSA-FDH-VRF\)](#)
  - [4.1. RSA-FDH-VRF Proving](#)
  - [4.2. RSA-FDH-VRF Proof to Hash](#)
  - [4.3. RSA-FDH-VRF Verifying](#)
  - [4.4. RSA-FDH-VRF Ciphersuites](#)
- [5. Elliptic Curve VRF \(ECVRF\)](#)
  - [5.1. ECVRF Proving](#)
  - [5.2. ECVRF Proof to Hash](#)
  - [5.3. ECVRF Verifying](#)
  - [5.4. ECVRF Auxiliary Functions](#)
    - [5.4.1. ECVRF Encode to Curve](#)
    - [5.4.2. ECVRF Nonce Generation](#)
    - [5.4.3. ECVRF Challenge Generation](#)
    - [5.4.4. ECVRF Decode Proof](#)
    - [5.4.5. ECVRF Validate Key](#)
  - [5.5. ECVRF Ciphersuites](#)
- [6. Implementation Status](#)
- [7. Security Considerations](#)
  - [7.1. Key Generation](#)
    - [7.1.1. Uniqueness and collision resistance under malicious key generation](#)
    - [7.1.2. Pseudorandomness under malicious key generation](#)
    - [7.1.3. Unpredictability under malicious key generation](#)
  - [7.2. Security Levels](#)
  - [7.3. Selective vs. Full Pseudorandomness](#)
  - [7.4. Proper pseudorandom nonce for ECVRF](#)
  - [7.5. Side-channel attacks](#)
  - [7.6. Proofs provide no secrecy for the VRF input](#)
  - [7.7. Prehashing](#)
  - [7.8. Hash function domain separation](#)
  - [7.9. Hash function salting](#)
  - [7.10. Futureproofing](#)
- [8. Change Log](#)
- [9. Contributors](#)
- [10. References](#)
  - [10.1. Normative References](#)
  - [10.2. Informative References](#)

## [Appendix A. Test Vectors for the RSA-FDH-VRF ciphersuites](#)

[A.1. RSA-FDH-VRF-SHA256](#)

[A.2. RSA-FDH-VRF-SHA384](#)

[A.3. RSA-FDH-VRF-SHA512](#)

## [Appendix B. Test Vectors for the ECVRF ciphersuites](#)

[B.1. ECVRF-P256-SHA256-TAI](#)

[B.2. ECVRF-P256-SHA256-SSWU](#)

[B.3. ECVRF-EDWARDS25519-SHA512-TAI](#)

[B.4. ECVRF-EDWARDS25519-SHA512-ELL2](#)

## [Authors' Addresses](#)

### **1. Introduction**

A Verifiable Random Function (VRF) [[MRV99](#)] is the public-key version of a keyed cryptographic hash. Only the holder of the VRF secret key can compute the hash, but anyone with the corresponding public key can verify the correctness of the hash.

A key application of the VRF is to provide privacy against offline dictionary attacks (also known as enumeration attacks) on data stored in a hash-based data structure. In this application, a Prover holds the VRF secret key and uses the VRF hashing to construct a hash-based data structure on the input data.

Due to the nature of the VRF, only the Prover can answer queries about whether or not some data is stored in the data structure. Anyone who knows the VRF public key can verify that the Prover has answered the queries correctly. However, no offline inferences (i.e. inferences without querying the Prover) can be made about the data stored in the data structure.

This document defines VRFs based on RSA and elliptic curves. The choices of VRFs for inclusion into this document were based, in part, on synergy with existing RFCs and commonly available implementations of individual components that are used within the VRFs.

The particular choice of the VRF for a given application depends on the desired security properties, the availability of cryptographically strong implementations, efficiency constraints, and the trust one places in RSA and elliptic curve Diffie-Hellman assumptions (and the trust in a particular choice of curve in case of elliptic curves). Differences in the security properties provided by the different options are discussed in [Section 3](#) and [Section 7](#).

This document represents the consensus of the Crypto Forum Research Group (CFRG).

#### **1.1. Requirements**

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [[RFC8174](#)].

#### **1.2. Terminology**

The following terminology is used through this document:

**SK:**

The secret key for the VRF. (Note: the secret key is also sometimes called "private key".)

**PK:** The public key for the VRF.

**alpha or alpha\_string:** The input to be hashed by the VRF.

**beta or beta\_string:** The VRF hash output.

**pi or pi\_string:** The VRF proof.

**Prover:** The Prover holds the VRF secret key SK and public key PK.

**Verifier:** The Verifier holds the VRF public key PK.

**Adversary:** Potential attacker; often used to define a security property.

**Malicious (or adversarial):** Performed by an adversary.

## 2. VRF Algorithms

A VRF comes with a key generation algorithm that generates a VRF public key PK and secret key SK.

The prover hashes an input alpha using the VRF secret key SK to obtain a VRF hash output beta

$$\text{beta} = \text{VRF\_hash}(\text{SK}, \text{alpha})$$

The VRF\_hash algorithm is deterministic, in the sense that it always produces the same output beta given the same pair of inputs (SK, alpha).

The prover also uses the secret key SK to construct a proof pi that beta is the correct hash output

$$\text{pi} = \text{VRF\_prove}(\text{SK}, \text{alpha})$$

The VRFs defined in this document allow anyone to deterministically obtain the VRF hash output beta directly from the proof value pi by using the function VRF\_proof\_to\_hash:

$$\text{beta} = \text{VRF\_proof\_to\_hash}(\text{pi})$$

Thus, for VRFs defined in this document, VRF\_hash is defined as

$$\text{VRF\_hash}(\text{SK}, \text{alpha}) = \text{VRF\_proof\_to\_hash}(\text{VRF\_prove}(\text{SK}, \text{alpha})),$$

and therefore this document will specify VRF\_prove and VRF\_proof\_to\_hash rather than VRF\_hash.

The proof pi allows a Verifier holding the public key PK to verify that beta is the correct VRF hash of input alpha under key PK. Thus, the VRFs defined in this document also come with an algorithm

$$\text{VRF\_verify}(\text{PK}, \text{alpha}, \text{pi})$$

that outputs (VALID,  $\beta = \text{VRF\_proof\_to\_hash}(\pi)$ ) if  $\pi$  is valid, and INVALID otherwise.

### 3. VRF Security Properties

VRFs are designed to ensure the following security properties: uniqueness (full or trusted), collision resistance (full or trusted), and pseudorandomness (full or selective). Some are designed to also ensure unpredictability under malicious key generation. We now describe these properties.

#### 3.1. Full Uniqueness

Uniqueness means that, for any fixed VRF public key and for any input  $\alpha$ , it is infeasible to find proofs for more than one VRF output  $\beta$ .

More precisely, "full uniqueness" means that an adversary cannot find

- \*a VRF public key PK,
- \*a VRF input  $\alpha$ ,
- \*and two proofs  $\pi_1$  and  $\pi_2$

such that

- \* $\text{VRF\_verify}(\text{PK}, \alpha, \pi_1)$  outputs (VALID,  $\beta_1$ ),
- \* $\text{VRF\_verify}(\text{PK}, \alpha, \pi_2)$  outputs (VALID,  $\beta_2$ ),
- \*and  $\beta_1$  is not equal to  $\beta_2$ .

#### 3.2. Full Collision Resistance

Like cryptographic hash functions, VRFs are collision resistant. Collision resistance means that it is infeasible to find two different inputs  $\alpha_1$  and  $\alpha_2$  with the same output  $\beta$ .

More precisely, "full collision resistance" means that an adversary cannot find

- \*a VRF public key PK,
- \*two VRF inputs  $\alpha_1$  and  $\alpha_2$  that are not equal to each other,
- \*and two proofs  $\pi_1$  and  $\pi_2$

such that

- \* $\text{VRF\_verify}(\text{PK}, \alpha_1, \pi_1)$  outputs (VALID,  $\beta_1$ ),
- \* $\text{VRF\_verify}(\text{PK}, \alpha_2, \pi_2)$  outputs (VALID,  $\beta_2$ ),
- \*and  $\beta_1$  is equal to  $\beta_2$ .

### 3.3. Trusted Uniqueness and Trusted Collision Resistance

Full uniqueness and full collision resistance hold even if the VRF keys are generated maliciously. For some applications, it is sufficient for a VRF to possess weaker security properties than full uniqueness and full collision resistance, called "trusted uniqueness" and "trusted collision resistance". These properties are the same as full uniqueness and full collision resistance, respectively, but are not guaranteed to hold if the adversary gets to choose the VRF public key PK. Instead, they are guaranteed to hold only if the VRF keys PK and SK are generated as specified by the VRF key generation algorithm and then given to the adversary. In other words, they are guaranteed to hold even if the adversary has the knowledge of SK and PK, but not guaranteed to hold if the adversary has the ability to choose SK and PK.

As further discussed in [Section 7.1.1](#), some VRFs specified in this document satisfy only trusted uniqueness and trusted collision resistance. VRFs in this document that satisfy only trusted uniqueness and trusted collision resistance MUST NOT be used in applications that need protection against adversarial VRF key generation.

### 3.4. Full Pseudorandomness or Selective Pseudorandomness

Pseudorandomness ensures that when someone who does not know SK sees a VRF hash output  $\beta$  without its corresponding VRF proof  $\pi$ , then  $\beta$  is indistinguishable from a random value.

More precisely, suppose the public and secret VRF keys (PK, SK) were generated correctly. Pseudorandomness ensures that the VRF hash output  $\beta$  (without its corresponding VRF proof  $\pi$ ) on any adversarially chosen "target" VRF input  $\alpha$  looks indistinguishable from random for any adversary who does not know the VRF secret key SK. This holds even if the adversary sees VRF hash outputs  $\beta'$  and proofs  $\pi'$  for multiple other inputs  $\alpha'$  (and even if those other inputs  $\alpha'$  are chosen by the adversary).

"Full pseudorandomness" security property holds even against an adversary who is allowed to choose the "target" VRF input  $\alpha$  at any time, even after it observes VRF outputs  $\beta'$  and proofs  $\pi'$  on a variety of chosen inputs  $\alpha'$ .

"Selective pseudorandomness" is a weaker security property that suffices in many applications. This security property holds against an adversary who chooses the target VRF input  $\alpha$  first, before it learns the VRF public key PK and obtains VRF outputs  $\beta'$  and proofs  $\pi'$  on other inputs  $\alpha'$  of its choice.

As further discussed in [Section 7.3](#), VRFs specified in this document satisfy both full pseudorandomness and selective pseudorandomness, but their quantitative security against the selective pseudorandomness attack is stronger.

It is important to remember that the VRF output  $\beta$  is always distinguishable from random by the Prover, or by any other party that knows the VRF secret key SK. Such a party can easily

distinguish beta from a random value by comparing beta to the result of `VRF_hash(SK, alpha)`.

Similarly, the VRF output beta is always distinguishable from random by any party that knows a valid VRF proof pi corresponding to the VRF input alpha, even if this party does not know the VRF secret key SK. Such a party can easily distinguish beta from a random value by checking whether `VRF_verify(PK, alpha, pi)` returns (VALID, beta).

Additionally, the VRF output beta may be distinguishable from random if VRF key generation was not done correctly. (For example, if VRF keys were generated with bad randomness.)

### 3.5. Unpredictability Under Malicious Key Generation

As explained in [Section 3.4](#), pseudorandomness cannot hold against malicious key generation. For instance, if an adversary outputs VRF keys that are deterministically generated (or hard-coded and publicly known), then the outputs are easily derived by anyone and are therefore not pseudorandom.

There is, however, a different type of unpredictability that is desirable in certain VRF applications (such as leader selection in the consensus protocols of [\[GHMVZ17\]](#) and [\[DGKR18\]](#)), called "unpredictability under malicious key generation". This property is similar to the unpredictability achieved by an (ordinary, unkeyed) cryptographic hash function: if the input has enough entropy (i.e., cannot be predicted), then the correct output is indistinguishable from uniformly random, no matter how the VRF keys are generated.

A formal definition of this property appears in Section 3.2 of [\[DGKR18\]](#). As further discussed in [Section 7.1.3](#), only some VRFs specified in this document satisfy this property.

## 4. RSA Full Domain Hash VRF (RSA-FDH-VRF)

The RSA Full Domain Hash VRF (RSA-FDH-VRF) is a VRF that, for suitable key lengths, satisfies the "trusted uniqueness", "trusted collision resistance", and "full pseudorandomness" properties defined in [Section 3](#), as further discussed in [Section 7](#). Its security follows from the standard RSA assumption in the random oracle model. Formal security proofs are in [\[PWHVNRG17\]](#).

The VRF computes the proof pi as a deterministic RSA signature on input alpha using the RSA Full Domain Hash Algorithm [\[RFC8017\]](#) parametrized with the selected hash algorithm. RSA signature verification is used to verify the correctness of the proof. The VRF hash output beta is simply obtained by hashing the proof pi with the selected hash algorithm.

The key pair for RSA-FDH-VRF MUST be generated in a way that it satisfies the conditions specified in Section 3 of [\[RFC8017\]](#).

In this section, the notation from [\[RFC8017\]](#) is used.

Parameters used:

(n, e) - RSA public key

K - RSA private key (its representation is implementation-dependent)

k - length in octets of the RSA modulus n (k must be less than  $2^{32}$ )

Fixed options (specified in [Section 4.4](#)):

Hash - cryptographic hash function

hLen - output length in octets of hash function Hash

suite\_string - an octet string specifying the RSA-FDH-VRF ciphersuite, which determines the above options

Primitives used:

I2OSP - Conversion of a nonnegative integer to an octet string as defined in Section 4.1 of [\[RFC8017\]](#) (given an integer and a length in octets, produces a big-endian representation of the integer, zero-padded to the desired length)

OS2IP - Conversion of an octet string to a nonnegative integer as defined in Section 4.2 of [\[RFC8017\]](#) (given a big-endian encoding of an integer, produces the integer)

RSASP1 - RSA signature primitive as defined in Section 5.2.1 of [\[RFC8017\]](#) (given a private key and an input, raises the input to the private RSA exponent modulo n)

RSASP1 - RSA verification primitive as defined in Section 5.2.2 of [\[RFC8017\]](#) (given a public key and an input, raises the input to the public RSA exponent modulo n)

MGF1 - Mask Generation Function based on the hash function Hash as defined in Section B.2.1 of [\[RFC8017\]](#) (given an input, produces a random-oracle-like output of desired length)

|| - octet string concatenation

#### 4.1. RSA-FDH-VRF Proving

RSADFHVRF\_prove(K, alpha\_string[, MGF\_salt])

Input:

K - RSA private key

alpha\_string - VRF hash input, an octet string

Optional Input:

MGF\_salt - a public octet string used as a hash function salt; this input is not used when MGF\_salt is specified as part of the ciphersuite



Output:

pi\_string - proof, an octet string of length k

Steps:

1. mgf\_domain\_separator = 0x01
2. EM = MGF1(suite\_string || mgf\_domain\_separator || MGF\_salt || alpha\_string, k - 1)
3. m = OS2IP(EM)
4. s = RSASP1(K, m)
5. pi\_string = I2OSP(s, k)
6. Output pi\_string

#### 4.2. RSA-FDH-VRF Proof to Hash

RSAFDHVRF\_proof\_to\_hash(pi\_string)

Input:

pi\_string - proof, an octet string of length k

Output:

beta\_string - VRF hash output, an octet string of length hLen

Important note:

RSAFDHVRF\_proof\_to\_hash should be run only on pi\_string that is known to have been produced by RSAFDHVRF\_prove, or from within RSAFDHVRF\_verify as specified in [Section 4.3](#).

Steps:

1. proof\_to\_hash\_domain\_separator = 0x02
2. beta\_string = Hash(suite\_string || proof\_to\_hash\_domain\_separator || pi\_string)
3. Output beta\_string

#### 4.3. RSA-FDH-VRF Verifying

RSAFDHVRF\_verify((n, e), alpha\_string, pi\_string[, MGF\_salt])

Input:

(n, e) - RSA public key

alpha\_string - VRF hash input, an octet string

pi\_string - proof to be verified, an octet string of length k

Optional Input:

MGF\_salt - a public octet string used as a hash function salt; this input is not used when MGF\_salt is specified as part of the ciphersuite

Output:

Output:

("VALID", beta\_string), where beta\_string is the VRF hash output, an octet string of length hLen; or

"INVALID"

Steps:

1.  $s = \text{OS2IP}(\text{pi\_string})$
2.  $m = \text{RSVP1}((n, e), s)$ ; if RSVP1 returns "signature representative out of range", output "INVALID" and stop.
3.  $\text{mgf\_domain\_separator} = 0x01$
4.  $\text{EM}' = \text{MGF1}(\text{suite\_string} || \text{mgf\_domain\_separator} || \text{MGF\_salt} || \text{alpha\_string}, k - 1)$
5.  $m' = \text{OS2IP}(\text{EM}')$
6. If  $m$  and  $m'$  are equal, output ("VALID",  $\text{RSVP1\_proof\_to\_hash}(\text{pi\_string})$ ); else output "INVALID".

#### 4.4. RSA-FDH-VRF Ciphersuites

This document defines RSA-FDH-VRF-SHA256 as follows:

\*suite\_string = 0x01

\*The hash function Hash is SHA-256 as specified in [\[RFC6234\]](#), with hLen = 32

\*MGF\_salt =  $\text{I2OSP}(k, 4) || \text{I2OSP}(n, k)$

This document defines RSA-FDH-VRF-SHA384 as follows:

\*suite\_string = 0x02

\*The hash function Hash is SHA-384 as specified in [\[RFC6234\]](#), with hLen = 48

\*MGF\_salt =  $\text{I2OSP}(k, 4) || \text{I2OSP}(n, k)$

This document defines RSA-FDH-VRF-SHA512 as follows:

\*suite\_string = 0x03

\*The hash function Hash is SHA-512 as specified in [\[RFC6234\]](#), with hLen = 64

\*MGF\_salt = I2OSP(k, 4) || I2OSP(n, k)

## 5. Elliptic Curve VRF (ECVRF)

The Elliptic Curve Verifiable Random Function (ECVRF) is a VRF that, for suitable parameter choices, satisfies the "full uniqueness", "trusted collision resistance", and "full pseudorandomness properties" defined in [Section 3](#). If `validate_key` parameter given to the `ECVRF_verify` is `TRUE`, then the ECVRF additionally satisfies "full collision resistance" and "unpredictability under malicious key generation". See [Section 7](#) for further discussion. Formal security proofs are in [[PWHVNRG17](#)].

Notation used:

Elliptic curve operations are written in additive notation, with  $P+Q$  denoting point addition and  $x*P$  denoting scalar multiplication of a point  $P$  by a scalar  $x$

$x^y$  -  $x$  raised to the power  $y$

$x*y$  -  $x$  multiplied by  $y$

$s || t$  - concatenation of octet strings  $s$  and  $t$

$0xMN$  (where  $M$  and  $N$  are hexadecimal digits) - a single octet with value  $M*16+N$ ; equivalently, `int_to_string(M*16+N, 1)`, where `int_to_string` is as defined below.

Fixed options (specified in [Section 5.5](#)):

$F$  - finite field

$fLen$  - length, in octets, of an element in  $F$  encoded as an octet string

$E$  - elliptic curve (EC) defined over  $F$

$ptLen$  - length, in octets, of a point on  $E$  encoded as an octet string

$G$  - subgroup of  $E$  of large prime order

$q$  - prime order of group  $G$

$qLen$  - length of  $q$  in octets, i.e., smallest integer such that  $2^{(8qLen)} > q$

$cLen$  - length, in octets, of a challenge value used by the VRF (note that in the typical case,  $cLen$  is  $qLen/2$  or close to it)

$cofactor$  - number of points on  $E$  divided by  $q$

$B$  - generator of group  $G$

Hash - cryptographic hash function

hLen - output length in octets of Hash (hLen must be at least cLen; in the typical case, it is at least qLen)

ECVRF\_encode\_to\_curve - a function that hashes strings to points on E.

ECVRF\_nonce\_generation - a function that derives a pseudorandom nonce from SK and the input as part of ECVRF proving.

suite\_string - an octet string specifying the ECVRF ciphersuite, which determines the above options as well as type conversions and parameter generation

Type conversions (specified in [Section 5.5](#)):

int\_to\_string(a, len) - conversion of nonnegative integer a to octet string of length len

string\_to\_int(a\_string) - conversion of an octet string a\_string to a nonnegative integer

point\_to\_string - conversion of a point on E to an ptLen-octet string

string\_to\_point - conversion of an ptLen-octet string to a point on E. string\_to\_point returns INVALID if the octet string does not convert to a valid EC point on the curve E.

Note that with certain software libraries (for big integer and elliptic curve arithmetic), the int\_to\_string and point\_to\_string conversions are not needed, when the libraries encode integers and EC points in the same way as required by the ciphersuites. For example, in some implementations, EC point operations will take octet strings as inputs and produce octet strings as outputs, without introducing a separate elliptic curve point type.

Parameters used (the generation of these parameters is specified in [Section 5.5](#)):

SK - VRF secret key

x - VRF secret scalar, an integer. Note: depending on the ciphersuite used, the VRF secret scalar may be equal to SK; else, it is derived from SK

$Y = x \cdot B$  - VRF public key, an point on E

PK\_string = point\_to\_string(Y) - VRF public key represented as an octet string

encode\_to\_curve\_salt - a public value used as a hash function salt

### 5.1. ECVRF Proving

ECVRF\_prove(SK, alpha\_string[, encode\_to\_curve\_salt])

Input:

SK - VRF secret key

alpha\_string - input alpha, an octet string

Optional input:

encode\_to\_curve\_salt - a public salt value, an octet string; this input is not used when encode\_to\_curve\_salt is specified as part of the ciphersuite

Output:

pi\_string - VRF proof, octet string of length ptLen+cLen+qLen

Steps:

1. Use SK to derive the VRF secret scalar  $x$  and the VRF public key  $Y = x*B$

(this derivation depends on the ciphersuite, as per [Section 5.5](#);

these values can be cached, for example, after key generation, and need not be rederived each time)

2.  $H = \text{ECVRF\_encode\_to\_curve}(\text{encode\_to\_curve\_salt}, \text{alpha\_string})$   
(see [Section 5.4.1](#))

3.  $h\_string = \text{point\_to\_string}(H)$

4.  $\text{Gamma} = x*H$

5.  $k = \text{ECVRF\_nonce\_generation}(SK, h\_string)$  (see [Section 5.4.2](#))

6.  $c = \text{ECVRF\_challenge\_generation}(Y, H, \text{Gamma}, k*B, k*H)$  (see [Section 5.4.3](#))

7.  $s = (k + c*x) \bmod q$

8.  $pi\_string = \text{point\_to\_string}(\text{Gamma}) || \text{int\_to\_string}(c, cLen) || \text{int\_to\_string}(s, qLen)$

9. Output pi\_string

## 5.2. ECVRF Proof to Hash

ECVRF\_proof\_to\_hash(pi\_string)

Input:

pi\_string - VRF proof, octet string of length ptLen+cLen+qLen

Output:

"INVALID", or

beta\_string - VRF hash output, octet string of length hLen

Important note:

ECVRF\_proof\_to\_hash should be run only on pi\_string that is known to have been produced by ECVRF\_prove, or from within ECVRF\_verify as specified in [Section 5.3](#).

Steps:

1. D = ECVRF\_decode\_proof(pi\_string) (see [Section 5.4.4](#))
2. If D is "INVALID", output "INVALID" and stop
3. (Gamma, c, s) = D
4. proof\_to\_hash\_domain\_separator\_front = 0x03
5. proof\_to\_hash\_domain\_separator\_back = 0x00
6. beta\_string = Hash(suite\_string ||  
proof\_to\_hash\_domain\_separator\_front ||  
point\_to\_string(cofactor \* Gamma) ||  
proof\_to\_hash\_domain\_separator\_back)
7. Output beta\_string

### 5.3. ECVRF Verifying

ECVRF\_verify(PK\_string, alpha\_string, pi\_string[,  
encode\_to\_curve\_salt, validate\_key])

Input:

PK\_string - public key, an octet string

alpha\_string - VRF input, octet string

pi\_string - VRF proof, octet string of length ptLen+cLen+qLen

Optional input:

encode\_to\_curve\_salt - a public salt value, an octet string; this input is not used when encode\_to\_curve\_salt is specified as part of the ciphersuite

validate\_key - a boolean. An implementation MAY support only the option of validate\_key = TRUE, or only the option of validate\_key = FALSE, in which case this input is not needed. If an implementation supports only one option, it MUST specify which option it supports.

Output:

("VALID", beta\_string), where beta\_string is the VRF hash output, octet string of length hLen; or

"INVALID"

Steps:

1.  $Y = \text{string\_to\_point}(\text{PK\_string})$
2. If  $Y$  is "INVALID", output "INVALID" and stop
3. If `validate_key`, run `ECVRF_validate_key(Y)` ([Section 5.4.5](#)); if it outputs "INVALID", output "INVALID" and stop
4.  $D = \text{ECVRF\_decode\_proof}(\text{pi\_string})$  (see [Section 5.4.4](#))
5. If  $D$  is "INVALID", output "INVALID" and stop
6.  $(\text{Gamma}, c, s) = D$
7.  $H = \text{ECVRF\_encode\_to\_curve}(\text{encode\_to\_curve\_salt}, \text{alpha\_string})$  (see [Section 5.4.1](#))
8.  $U = s*B - c*Y$
9.  $V = s*H - c*\text{Gamma}$
10.  $c' = \text{ECVRF\_challenge\_generation}(Y, H, \text{Gamma}, U, V)$  (see [Section 5.4.3](#))
11. If  $c$  and  $c'$  are equal, output ("VALID", `ECVRF_proof_to_hash(pi_string)`); else output "INVALID"

Note that the first three steps need to be performed only once for a given public key.

## 5.4. ECVRF Auxiliary Functions

### 5.4.1. ECVRF Encode to Curve

The `ECVRF_encode_to_curve` algorithm takes a public salt (see [Section 7.9](#)) and the VRF input `alpha` and converts it to  $H$ , an EC point in  $G$ . This algorithm is the only place the VRF input `alpha` is used for proving and verifying. See [Section 7.7](#) for further discussion.

This section specifies a number of such algorithms, which are not compatible with each other and are intended to use with various ciphersuites specified in [Section 5.5](#).

Input:

`encode_to_curve_salt` - public salt value, an octet string

`alpha_string` - value to be hashed, an octet string

Output:

$H$  - hashed value, a point in  $G$

#### 5.4.1.1. ECVRF\_encode\_to\_curve\_try\_and\_increment

The following

`ECVRF_encode_to_curve_try_and_increment(encode_to_curve_salt,`

alpha\_string) algorithm implements ECVRF\_encode\_to\_curve in a simple and generic way that works for any elliptic curve. To use this algorithm, hLen MUST be at least fLen.

The running time of this algorithm depends on alpha\_string. For the ciphersuites specified in [Section 5.5](#), this algorithm is expected to find a valid curve point after approximately two attempts (i.e., when ctr=1) on average.

However, because the running time of algorithm depends on alpha\_string, this algorithm SHOULD be avoided in applications where it is important that the VRF input alpha remain secret.

```
ECVRF_encode_to_curve_try_and_increment(encode_to_curve_salt,  
alpha_string)
```

Fixed option (specified in [Section 5.5](#)):

interpret\_hash\_value\_as\_a\_point - a function that attempts to convert a cryptographic hash value to a point on E; may output INVALID.

Steps:

1. ctr = 0
2. encode\_to\_curve\_domain\_separator\_front = 0x01
3. encode\_to\_curve\_domain\_separator\_back = 0x00
4. H = "INVALID"
5. While H is "INVALID" or H is the identity element of the elliptic curve group:
  - a. ctr\_string = int\_to\_string(ctr, 1)
  - b. hash\_string = Hash(suite\_string ||  
encode\_to\_curve\_domain\_separator\_front ||  
encode\_to\_curve\_salt || alpha\_string || ctr\_string ||  
encode\_to\_curve\_domain\_separator\_back)
  - c. H = interpret\_hash\_value\_as\_a\_point(hash\_string)
  - d. If H is not "INVALID" and cofactor > 1, set H = cofactor \*  
H
  - e. ctr = ctr + 1
6. Output H

Note even though the loop is infinite as written, and int\_to\_string(ctr,1) may fail when ctr reaches 256, interpret\_hash\_value\_as\_a\_point functions specified in [Section 5.5](#) will succeed on roughly half hash\_string values. Thus the loop is expected to stop after two iterations, and ctr is overwhelmingly unlikely (probability about  $2^{-256}$ ) to reach 256.



#### 5.4.1.2. ECVRF\_encode\_to\_curve\_h2c\_suite

The `ECVRF_encode_to_curve_h2c_suite(encode_to_curve_salt, alpha_string)` algorithm implements `ECVRF_encode_to_curve` using one of the several hash-to-curve options defined in [[I-D.irtf-cfrg-hash-to-curve](#)]. The specific choice of the hash-to-curve option (called Suite ID in [[I-D.irtf-cfrg-hash-to-curve](#)]) is given by the `h2c_suite_ID_string` parameter.

`ECVRF_encode_to_curve_h2c_suite(encode_to_curve_salt, alpha_string)`

Fixed option (specified in [Section 5.5](#)):

`h2c_suite_ID_string` - a hash-to-curve suite ID, encoded in ASCII (see discussion below)

Steps:

1. `string_to_be_hashed = encode_to_curve_salt || alpha_string`
2. `H = encode(string_to_be_hashed)`  
(the encode function is discussed below)
3. Output H

The encode function is provided by the hash-to-curve suite whose ID is `h2c_suite_ID_string`, as specified in [[I-D.irtf-cfrg-hash-to-curve](#)], Section 8. The domain separation tag DST, a parameter to the hash-to-curve suite, SHALL be set to

`"ECVRF_" || h2c_suite_ID_string || suite_string`

where "ECVRF\_" is represented as a 6-byte ASCII encoding (in hexadecimal, octets 45 43 56 52 46 5F).

#### 5.4.2. ECVRF Nonce Generation

The following algorithms generate the nonce value `k` in a deterministic pseudorandom fashion. This section specifies a number of such algorithms, which are not compatible with each other. The choice of a particular algorithm from the options specified in this section depends on the ciphersuite, as specified in [Section 5.5](#).

##### 5.4.2.1. ECVRF Nonce Generation from RFC 6979

`ECVRF_nonce_generation_RFC6979(SK, h_string)`

Input:

`SK` - an ECVRF secret key

`h_string` - an octet string

Output:

`k` - an integer nonce between 1 and `q-1`

The ECVRF\_nonce\_generation function is as specified in [[RFC6979](#)] Section 3.2 where

Input  $m$  is set equal to  $h\_string$

The "suitable for DSA or ECDSA" check in step h.3 is omitted

The hash function  $H$  is Hash and its output length  $hlen$  (in bits) is set as  $hlen*8$

The secret key  $x$  is set equal to the VRF secret scalar  $x$

The prime  $q$  is the same as in this specification

$qlen$  is the binary length of  $q$ , i.e., the smallest integer such that  $2^{qlen} > q$  (this  $qlen$  is not to be confused with  $qLen$  in this document, which is the length of  $q$  in octets)

All the other values and primitives as defined in [[RFC6979](#)]

#### 5.4.2.2. ECVRF Nonce Generation from RFC 8032

The following is from Steps 2-3 of Section 5.1.6 in [[RFC8032](#)]. To use this algorithm,  $hlen$  MUST be at least 64.

ECVRF\_nonce\_generation\_RFC8032(SK,  $h\_string$ )

Input:

SK - an ECVRF secret key

$h\_string$  - an octet string

Output:

$k$  - an integer nonce between 0 and  $q-1$

Steps:

1.  $hashed\_sk\_string = Hash(SK)$
2.  $truncated\_hashed\_sk\_string = hashed\_sk\_string[32] \dots hashed\_sk\_string[63]$
3.  $k\_string = Hash(truncated\_hashed\_sk\_string || h\_string)$
4.  $k = string\_to\_int(k\_string) \bmod q$

#### 5.4.3. ECVRF Challenge Generation

ECVRF\_challenge\_generation( $P1, P2, P3, P4, P5$ )

Input:

$P1, P2, P3, P4, P5$  - EC points

Output:

c - challenge value, integer between 0 and  $2^{(8*cLen)}-1$

Steps:

1. challenge\_generation\_domain\_separator\_front = 0x02
2. Initialize str = suite\_string || challenge\_generation\_domain\_separator\_front
3. for PJ in [P1, P2, P3, P4, P5]:  
    str = str || point\_to\_string(PJ)
4. challenge\_generation\_domain\_separator\_back = 0x00
5. str = str || challenge\_generation\_domain\_separator\_back
6. c\_string = Hash(str)
7. truncated\_c\_string = c\_string[0]...c\_string[cLen-1]
8. c = string\_to\_int(truncated\_c\_string)
9. Output c

#### 5.4.4. ECVRF Decode Proof

ECVRF\_decode\_proof(pi\_string)

Input:

pi\_string - VRF proof, octet string (ptLen+cLen+qLen octets)

Output:

"INVALID", or

Gamma - a point on E

c - integer between 0 and  $2^{(8*cLen)}-1$

s - integer between 0 and q-1

Steps:

1. gamma\_string = pi\_string[0]...pi\_string[ptLen-1]
2. c\_string = pi\_string[ptLen]...pi\_string[ptLen+cLen-1]
3. s\_string = pi\_string[ptLen+cLen]...pi\_string[ptLen+cLen+qLen-1]
4. Gamma = string\_to\_point(gamma\_string)
5. if Gamma = "INVALID" output "INVALID" and stop
6. c = string\_to\_int(c\_string)

7. `s = string_to_int(s_string)`
8. if `s >= q` output "INVALID" and stop
9. Output Gamma, c, and s

#### 5.4.5. ECVRF Validate Key

`ECVRF_validate_key(Y)`

Input:

Y - public key, a point on E

Output:

"VALID" or "INVALID"

Important note: the public key Y given to this procedure MUST be a valid point on E.

Steps:

1. Let  $Y' = \text{cofactor} * Y$
2. If  $Y'$  is the identity element of the elliptic curve group, output "INVALID" and stop
3. Output "VALID"

Note that if the cofactor = 1, then Step 1 simply sets  $Y'=Y$ . In particular, for the P-256 curve, `ECVRF_validate_key` simply ensures that Y is not the point at infinity.

Any algorithm with identical input-output behavior MAY be used in place of the above steps. For example, if the total number of Y values that could cause Step 2 to output "INVALID" is small, it may be more efficient to simply check Y against a fixed list of such values. For example, the following algorithm MAY be used for the `edwards25519` curve:

1. `PK_string = point_to_string(Y)`
2. `oneTwentySeven_string = 0x7F`
3. `y_string[31] = y_string[31] & oneTwentySeven_string`  
(this step clears the high-order bit of octet 31)
4. `bad_pk[0] = int_to_string(0, 32)`
5. `bad_pk[1] = int_to_string(1, 32)`
6. `bad_y2 =`  
270738550114484064931822528722565878893680426757531351946374360  
9750303402022
7. `bad_pk[2] = int_to_string(bad_y2, 32)`

8. `bad_pk[3] = int_to_string(p-bad_y2, 32)`
9. `bad_pk[4] = int_to_string(p-1, 32)`
10. `bad_pk[5] = int_to_string(p, 32)`
11. `bad_pk[6] = int_to_string(p+1, 32)`
12. If `y_string` is in the list `[bad_pk[0],...,bad_pk[6]]`, output "INVALID" and stop
13. Output "VALID"

(This algorithm works for the following reason. Note that there are 8 bad points -- namely, the points whose order is 1, 2, 4, or 8 -- on the edwards25519 curve. Their y coordinates happen to be 0 (two points of order 4), 1 (one point of order 1), `bad_y2` (two points of order 8), `p-bad_y2` (two points of order 8), and `p-1` (one point of order 2). They can be obtained by converting the points specified in [\[X25519\]](#) to Edwards coordinates. Thus, `bad_pk[0]` (of order 4), `bad_pk[2]` (of order 8), and `bad_pk[3]` (of order 8) each match two bad points, depending on the sign of the x-coordinate, which was cleared in step 3, in order to make sure that it does not affect the comparison. `bad_pk[1]` (of order 1) and `bad_pk[4]` (of order 2) each match one bad point, because x-coordinate is 0 for these two points. Note that the first 5 list elements cover the 8 bad points. However, in case the y-coordinate of the public key Y had not been modular reduced by p, the list also includes `bad_pk[5]` and `bad_pk[6]`, which are simply `bad_pk[0]` and `bad_pk[1]` shifted by p. There is no need to shift the other `bad_pk` values by p (or any `bad_pk` values by a larger multiple of p), because their y coordinate would exceed  $2^{255}$ ; and we ensure that `y_string` corresponds to an integer less than  $2^{255}$  in step 3.)

## 5.5. ECVRF Ciphersuites

This document defines ECVRF-P256-SHA256-TAI as follows:

\*suite\_string = 0x01.

\*The EC group G is the NIST P-256 elliptic curve, with curve parameters as specified in [\[FIPS-186-4\]](#) (Section D.1.2.3) and [\[RFC5114\]](#) (Section 2.6). For this group, `fLen = qLen = 32` and `cofactor = 1`.

\*cLen = 16.

\*The key pair generation primitive is specified in Section 3.2.1 of [\[SECG1\]](#) (`q`, `B`, `SK`, and `Y` in this document correspond to `n`, `G`, `d`, and `Q` in Section 3.2.1 of [\[SECG1\]](#)). In this ciphersuite, the secret scalar `x` is equal to the secret key `SK`.

\*encode\_to\_curve\_salt = PK\_string

\*The ECVRF\_nonce\_generation function is as specified in [Section 5.4.2.1](#).

\*The `int_to_string` function is the I2OSP function specified in Section 4.1 of [\[RFC8017\]](#). (This is big-endian representation.)

\*The `string_to_int` function is the OS2IP function specified in Section 4.2 of [\[RFC8017\]](#). (This is big-endian representation.)

\*The `point_to_string` function converts a point on E to an octet string according to the encoding specified in Section 2.3.3 of [\[SECG1\]](#) with point compression on. This implies `ptLen = fLen + 1 = 33`. (Note that certain software implementations do not introduce a separate elliptic curve point type and instead directly treat the EC point as an octet string per above encoding. When using such an implementation, the `point_to_string` function can be treated as the identity function.)

\*The `string_to_point` function converts an octet string to a point on E according to the encoding specified in Section 2.3.4 of [\[SECG1\]](#). This function MUST output INVALID if the octet string does not decode to a point on the curve E.

\*The hash function Hash is SHA-256 as specified in [\[RFC6234\]](#), with `hLen = 32`.

\*The `ECVRF_encode_to_curve` function is as specified in [Section 5.4.1.1](#), with `interpret_hash_value_as_a_point(s) = string_to_point(0x02 || s)`.

This document defines ECVRF-P256-SHA256-SSWU as identical to ECVRF-P256-SHA256-TAI, except that:

\*`suite_string = 0x02`.

\*the `ECVRF_encode_to_curve` function is as specified in [Section 5.4.1.2](#) with `h2c_suite_ID_string = P256_XMD:SHA-256_SSWU_NU` (the suite is defined in [\[I-D.irtf-cfrg-hash-to-curve\]](#) Section 8.2)

This document defines ECVRF-EDWARDS25519-SHA512-TAI as follows:

\*`suite_string = 0x03`.

\*The EC group G is the edwards25519 elliptic curve with parameters defined in Table 1 of [\[RFC8032\]](#). For this group, `fLen = qLen = 32` and `cofactor = 8`.

\*`cLen = 16`.

\*The secret key and generation of the secret scalar and the public key are specified in Section 5.1.5 of [\[RFC8032\]](#).

\*`encode_to_curve_salt = PK_string`

\*The `ECVRF_nonce_generation` function is as specified in [Section 5.4.2.2](#).

\*The `int_to_string` function as specified in the first paragraph of Section 5.1.2 of [\[RFC8032\]](#). (This is little-endian representation.)

\*The `string_to_int` function interprets the string as an integer in little-endian representation.

\*The `point_to_string` function converts a point on E to an octet string according to the encoding specified in Section 5.1.2 of [RFC8032]. This implies `ptLen = fLen = 32`. (Note that certain software implementations do not introduce a separate elliptic curve point type and instead directly treat the EC point as an octet string per above encoding. When using such an implementation, the `point_to_string` function can be treated as the identity function.)

\*The `string_to_point` function converts an octet string to a point on E according to the encoding specified in Section 5.1.3 of [RFC8032]. This function MUST output INVALID if the octet string does not decode to a point on the curve E.

\*The hash function Hash is SHA-512 as specified in [RFC6234], with `hLen = 64`.

\*The `ECVRF_encode_to_curve` function is as specified in Section 5.4.1.1, with `interpret_hash_value_as_a_point(s) = string_to_point(s[0]...s[31])`.

This document defines ECVRF-EDWARDS25519-SHA512-ELL2 as identical to ECVRF-EDWARDS25519-SHA512-TAI, except:

\*`suite_string = 0x04`.

\*the `ECVRF_encode_to_curve` function is as specified in Section 5.4.1.2 with `h2c_suite_ID_string = edwards25519_XMD:SHA-512_ELL2_NU_` (the suite is defined in [I-D.irtf-cfrg-hash-to-curve] Section 8.5).

## 6. Implementation Status

Note to RFC editor: Remove before publication

A reference C++ implementation of ECVRF-P256-SHA256-TAI, ECVRF-P256-SHA256-SSWU, ECVRF-EDWARDS25519-SHA512-TAI, and ECVRF-EDWARDS25519-SHA512-ELL2 is available at <https://github.com/reyzin/ecvrf>. This implementation is neither secure nor especially efficient, but can be used to generate test vectors.

A Python implementation of an older version of ECVRF-EDWARDS25519-SHA512-ELL2 from the -05 version of this draft is available at <https://github.com/integritychain/draft-irtf-cfrg-vrf-05>.

A C implementation of an older version of ECVRF-EDWARDS25519-SHA512-ELL2 from the -03 version of this draft is available at [https://github.com/algorand/libsodium/tree/draft-irtf-cfrg-vrf-03/src/libsodium/crypto\\_vrf/ietfdraft03](https://github.com/algorand/libsodium/tree/draft-irtf-cfrg-vrf-03/src/libsodium/crypto_vrf/ietfdraft03).

A Rust implementation of an older version of ECVRF-P256-SHA256-TAI from the -05 version of this draft, as well as variants for the `sect163k1` and `secp256k1` curves, is available at <https://crates.io/crates/vrf>.

A C implementation of a variant of ECVRF-P256-SHA256-TAI from the -05 version of this draft adapted for the secp256k1 curve is available at <https://github.com/aergoio/secp256k1-vrf>.

An implementation of an earlier version of RSA-FDH-VRF (SHA-256) and ECVRF-P256-SHA256-TAI was first developed as a part of the NSEC5 project [I-D.vcelak-nsec5] and is available at <http://github.com/fcelda/nsec5-crypto>.

The Key Transparency project at Google uses a VRF implementation that is similar to the ECVRF-P256-SHA256-TAI, with a few changes including the use of SHA-512 instead of SHA-256. Its implementation is available at <https://github.com/google/keytransparency/blob/master/core/crypto/vrf/>

An implementation by Ryuji Ishiguro following an older version of ECVRF-EDWARDS25519-SHA512-TAI from the -00 version of this draft is available at <https://github.com/r2ishiguro/vrf>.

An implementation similar to ECVRF-EDWARDS25519-SHA512-ELL2 (with some changes, including the use of SHA-3) is available as part of the CONIKS implementation in Golang at <https://github.com/coniks-sys/coniks-go/tree/master/crypto/vrf>.

Open Whisper Systems also uses a VRF similar to ECVRF-EDWARDS25519-SHA512-ELL2, called VXEdDSA, and specified here <https://whispersystems.org/docs/specifications/xeddsa/> and here <https://moderncrypto.org/mail-archive/curves/2017/000925.html>. Implementations in C and Java are available at <https://github.com/signalapp/curve25519-java> and <https://github.com/wavesplatform/curve25519-java>.

## 7. Security Considerations

### 7.1. Key Generation

Implementations of VRFs defined in this document MUST ensure that they generate VRF keys correctly and using good randomness. However, in some applications keys may be generated by an adversary who does not necessarily implement this document. We now discuss the implications of this possibility.

#### 7.1.1. Uniqueness and collision resistance under malicious key generation

See [Section 3](#) for definitions of uniqueness and collision resistance properties.

The RSA-FDH-VRF satisfies only the "trusted" variants of uniqueness and collision resistance. Thus, for RSA-FDH-VRF, uniqueness and collision resistance may not hold if the keys are generated adversarially (specifically, if the RSA function specified in the public key is not bijective because the modulus  $n$  or the exponent  $e$  are chosen not in compliance with [\[RFC8017\]](#)); thus, RSA-FDH-VRF defined in this document does not have "full uniqueness" and "full collision resistance". Therefore, if malicious key generation is a concern, the RSA-FDH-VRF has to be enhanced by additional cryptographic checks (such as zero-knowledge proofs) that its public



key has the right form. These enhancements are left for future specification.

For the ECVRF, the Verifier MUST obtain E and B from a trusted source, such as a ciphersuite specification, rather than from the prover. If the verifier does so, then the ECVRF satisfies the "full uniqueness", ensuring uniqueness even under malicious key generation. The ECVRF also satisfies "trusted collision resistance". It additionally satisfies "full collision resistance" if `validate_key` parameter given to the `ECVRF_verify` is TRUE. This setting of `ECVRF_verify` ensures collision resistance under malicious key generation.

### 7.1.2. Pseudorandomness under malicious key generation

Without good randomness, the "pseudorandomness" properties of the VRF (defined in [Section 3.4](#)) may not hold. Note that it is not possible to guarantee pseudorandomness in the face of adversarially generated VRF keys. This is because an adversary can always use bad randomness to generate the VRF keys, and thus, the VRF output may not be pseudorandom.

### 7.1.3. Unpredictability under malicious key generation

Unpredictability under malicious key generation (defined in [Section 3.5](#)) does not hold for the RSA-FDH-VRF. (Specifically, the VRF output may be predictable if the RSA function specified in the public key is far from bijective because the modulus  $n$  or the exponent  $e$  are chosen not in compliance with [\[RFC8017\]](#).) If unpredictability under malicious key generation is desired, the RSA-FDH-VRF has to be enhanced by additional cryptographic checks (such as zero-knowledge proofs) that its public key has the right form. These enhancements are left for future specification.

Unpredictability under malicious key generation holds for the ECVRF if `validate_key` parameter given to the `ECVRF_verify` is TRUE.

## 7.2. Security Levels

As shown in [\[PWHVNRG17\]](#), RSA-FDH-VRF satisfies the trusted uniqueness property unconditionally. The security level of the RSA-FDH-VRF, measured in bits, for the other two properties is as follows (in the random oracle model for the functions MGF1 and Hash):

- \*For trusted collision resistance: approximately  $8 \cdot \min(k/2, hLen/2)$  (as shown in [\[PWHVNRG17\]](#)).

- \*For selective pseudorandomness: approximately as strong as the security, in bits, of the RSA problem for the key  $(n, e)$  (as shown in [\[GNPRVZ15\]](#)).

As shown in [\[PWHVNRG17\]](#), the security level of the ECVRF, measured in bits, is as follows (in the random oracle model for the functions Hash and `ECVRF_encode_to_curve`):

- \*For uniqueness (both trusted and full): approximately  $8 \cdot \min(qLen, cLen)$ .

\*For collision resistance (trusted or full, depending on whether validation is performed as explained in [Section 7.1.1](#)): approximately  $8 \cdot \min(qLen/2, hLen/2)$ .

\*For the selective pseudorandomness property: approximately as strong as the security, in bits, of the decisional Diffie-Hellman problem in the group  $G$  (which is at most  $8 \cdot qLen/2$ ).

See [Section 3](#) for the definitions of these security properties. See [Section 7.3](#) for the discussion of full pseudorandomness.

### 7.3. Selective vs. Full Pseudorandomness

[[PWHVNRG17](#)] presents cryptographic reductions to an underlying hard problem (namely, the RSA problem for RSA-FDH-VRF and the Decisional Diffie-Hellman problem for the ECVRF) to prove that the VRFs specified in this document possess not only selective pseudorandomness, but also full pseudorandomness (see [Section 3.4](#) for an explanation of these notions). However, the cryptographic reductions are tighter for selective pseudorandomness than for full pseudorandomness. Specifically, the approximate provable security level, measured in bits, for full pseudorandomness may be obtained from the provable security level for selective pseudorandomness (given in [Section 7.2](#)) by subtracting the binary logarithm of the number of proofs produced for a given secret key. This holds for both the RSA-FDH-VRF and the ECVRF.

While no known attacks against full pseudorandomness are stronger than similar attacks against selective pseudorandomness, some applications may be concerned about tightness of cryptographic reductions to ensure specific levels of provable security. Such applications may consider the following three options:

\*They may limit the number of proofs produced for a given secret key, to reduce the loss in the provable security level.

\*They may work to ensure that selective pseudorandomness is sufficient for the application. That is, they may design the application in such a way that pseudorandomness of outputs matters only for inputs that are chosen independently of the VRF key.

\*They may increase security parameters to make up for the loose security reduction. For RSA-FDH-VRF, this means increasing the RSA key length. For ECVRF, this means increasing the cryptographic strength of the EC group  $G$  by specifying a new ciphersuite.

### 7.4. Proper pseudorandom nonce for ECVRF

The security of the ECVRF defined in this document relies on the fact that the nonce  $k$  used in the ECVRF\_prove algorithm is chosen uniformly and pseudorandomly modulo  $q$ , and is unknown to the adversary. Otherwise, an adversary may be able to recover the VRF secret scalar  $x$  (and thus break pseudorandomness of the VRF) after observing several valid VRF proofs  $\pi_i$ , using, for example, techniques described in [[BreHen19](#)]. The nonce generation methods

specified in the ECVRF ciphersuites of [Section 5.5](#) are designed with this requirement in mind.

## 7.5. Side-channel attacks

Side channel attacks on cryptographic primitives are an important issue. Implementers should take care to avoid side-channel attacks that leak information about the VRF secret key SK (and the nonce  $k$  used in the ECVRF), which is used in VRF\_prove. In most applications, VRF\_proof\_to\_hash and VRF\_verify algorithms take only inputs that are public, and thus side channel attacks are typically not a concern for these algorithms.

The VRF input alpha may be also a sensitive input to VRF\_prove and may need to be protected against side channel attacks. Below we discuss one particular class of such attacks: timing attacks that can be used to leak information about the VRF input alpha.

The ECVRF\_encode\_to\_curve\_try\_and\_increment algorithm defined in [Section 5.4.1.1](#) SHOULD NOT be used in applications where the VRF input alpha is secret and is hashed by the VRF on-the-fly. This is because the algorithm's running time depends on the VRF input alpha, and thus creates a timing channel that can be used to learn information about alpha. That said, for most inputs the amount of information obtained from such a timing attack is likely to be small (1 bit, on average), since the algorithm is expected to find a valid curve point after only two attempts. However, there might be inputs which cause the algorithm to make many attempts before it finds a valid curve point; for such inputs, the information leaked in a timing attack will be more than 1 bit.

ECVRF-P256-SHA256-SSWU and ECVRF-EDWARDS25519-SHA512-ELL2 can be made to run in time independent of alpha, following recommendations in [\[I-D.irtf-cfrg-hash-to-curve\]](#).

## 7.6. Proofs provide no secrecy for the VRF input

The VRF proof  $\pi$  is not designed to provide secrecy and, in general, may reveal the VRF input alpha. Anyone who knows PK and  $\pi$  is able to perform an offline dictionary attack to search for alpha, by verifying guesses for alpha using VRF\_verify. This is in contrast to the VRF hash output beta which, without the proof, is pseudorandom and thus is designed to reveal no information about alpha.

## 7.7. Prehashing

The VRFs specified in this document allow for read-once access to the input alpha for both signing and verifying. Thus, additional prehashing of alpha (as specified, for example, in [\[RFC8032\]](#) for EdDSA signatures) is not needed, even for applications that need to handle long alpha or to support the Initialize-Update-Finalize (IUF) interface (in such an interface, alpha is not supplied all at once, but rather in pieces by a sequence of calls to Update). The ECVRF, in particular, uses alpha only in ECVRF\_encode\_to\_curve. The curve point  $H$  becomes the representative of alpha thereafter.

## 7.8. Hash function domain separation

Hashing is used for different purposes in the two VRFs. Specifically, in the RSA-FDH-VRF, hashing is used in MGF1 and in `proof_to_hash`; in the ECVRF, hashing is used in `encode_to_curve`, `nonce_generation`, `challenge_generation`, and `proof_to_hash`. The theoretical analysis treats each of these functions as a separate hash function, modeled as a random oracle. This analysis still holds even if the same hash function is used, as long as the four inputs given to the hash function for a given SK and alpha are overwhelmingly unlikely to equal each other or to any inputs given to the hash function for the same SK and different alpha. This is indeed the case for the RSA-FDH-VRF defined in this document, because the second octets of the input to the hash function used in MGF1 and in `proof_to_hash` are different.

This is also the case for the ECVRF ciphersuites defined in this document, because:

- \*inputs to the hash function used during `nonce_generation` are unlikely to equal inputs used in `encode_to_curve`, `proof_to_hash`, and `challenge_generation`. This follows since `nonce_generation` inputs a secret to the hash function that is not used by honest parties as input to any other hash function, and is not available to the adversary.

- \*the second octets of the inputs to the hash function used in `proof_to_hash`, `challenge_generation`, and `encode_to_curve_try_and_increment` are all different.

- \*the last octet of the input to the hash function used in `proof_to_hash`, `challenge_generation`, and `encode_to_curve_try_and_increment` is always zero, and therefore different from the last octet of the input to the hash function used in `ECVRF_encode_to_curve_h2c_suite`, which is set equal to the nonzero length of the domain separation tag by [[I-D.irtf-cfrg-hash-to-curve](#)].

## 7.9. Hash function salting

In case a hash collision is found, in order to make it more difficult for the adversary to exploit such a collision, the MGF1 function for the RSA-FDH-VRF and `ECVRF_encode_to_curve` function for the ECVRF use a public value in addition to alpha (as a so-called salt). This value is determined by the ciphersuite. For the ciphersuites defined in this document, it is set equal to the string representation of the RSA modulus and EC public key, respectively. Implementations that do not use one of the ciphersuites (see [Section 7.10](#)) MAY use a different salt. For example, if a group of public keys to share the same salt, then the hash of the VRF input alpha will be the same for the entire group of public keys, which may aid in some protocol that uses the VRF.

## 7.10. Futureproofing

If future designs need to specify variants (e.g., additional ciphersuites) of the RSA-FDH-VRF or the ECVRF in this document, then, to avoid the possibility that an adversary can obtain a VRF

output under one variant, and then claim it was obtained under another variant, they should specify a different `suite_string` constant. The `suite_string` constants in this document are all single octets; if a future `suite_string` constant is longer than one octet, then it should start with a different octet than the `suite_string` constants in this document. Then, for the RSA-FDH-VRF, the inputs to the hash function used in MGF1 and `proof_to_hash` will be different from other ciphersuites. For the ECVRF, the inputs `ECVRF_encode_to_curve` hash function used in producing H are then guaranteed to be different from other ciphersuites; since all the other hashing done by the prover depends on H, inputs to all the hash functions used by the prover will also be different from other ciphersuites as long as `ECVRF_encode_to_curve` is collision resistant.

## 8. Change Log

Note to RFC Editor: if this document does not obsolete an existing RFC, please remove this appendix before publication as an RFC.

00 - Forked this document from draft-goldbe-vrf-01.

01 - Minor updates, mostly highlighting TODO items.

02 - Added specification of `elligator2` for Curve25519, along with ciphersuites for ECVRF-ED25519-SHA512-Elligator. Changed ECVRF-ED25519-SHA256 `suite_string` to ECVRF-ED25519-SHA512. (This change made because Ed25519 in [\[RFC8032\]](#) signatures use SHA512 and not SHA256.) Made ECVRF nonce generation a separate component, so that nonces are deterministic. In ECVRF proving, changed + to - (and made corresponding verification changes) in order to be consistent with EdDSA and ECDSA. Highlighted that `ECVRF_hash_to_curve` acts like a prehash. Added "suites" variable to ECVRF for futureproofing. Ensured domain separation for hash functions by modifying `hash_points` and added discussion about domain separation. Updated todos in the "additional pseudorandomness property" section. Added a discussion of secrecy into security considerations. Removed B and PK=Y from `ECVRF_hash_points` because they are already present via H, which is computed via `hash_to_curve` using the `suite_string` (which identifies B) and Y.

03 - Changed Ed25519 conversions to little-endian, to match RFC 8032; added simple key validation for Ed25519; added Simple SWU cipher suite; clarified Elligator and removed the extra x0 bit, to make Montgomery and Edwards Elligator the same; added domain separation for RSA VRF; improved notation throughout; added nonce generation as a section; changed counter in try-and-increment from four bytes to one, to avoid endian issues; renamed try-and-increment ciphersuites to -TAI; added `qLen` as a separate parameter; changed output length to `hLen` for ECVRF, to match RSAVRF; made `Verify` return beta so unverified proofs don't end up in `proof_to_hash`; added test vectors.

04 - Clarified handling of optional arguments `x` and `PK` in `ECVRF_prove`. Edited implementation status to bring it up to date.

05 - Renamed ed25519 into the more commonly used edwards25519. Corrected ECVRF\_nonce\_generation\_RFC6979 (thanks to Gorka Irazoqui Apecechea and Mario Cao Cueto for finding the problem) and corresponding test vectors for the P256 suites. Added a reference to the Rust implementation.

06 - Made some variable names more descriptive. Added a few implementation references.

07 - Incorporated hash-to-curve draft by reference to replace our own Elligator2 and Simple SWU. Clarified discussion of EC parameters and functions. Added a 0 octet to all hashing to enforce domain separation from hashing done inside hash-to-curve.

08 - Incorporated suggestions from crypto panel review by Chloe Martindale. Changed Reyzin's affiliation. Updated references.

09 - Added a note to remove the implementation page before publication.

10 - Added a check in ECVRF\_decode\_proof to ensure that  $s$  is reduced mod  $q$ . Connected security properties (Section 3) and security considerations (Section 7) with more cross-references.

11 - Processed last call comments. Clarified various notation, including lengths of various parameters for ECVRF; added error handling to RSA-FDH-VRF; added security levels section; clarified full vs trusted uniqueness and full vs selective pseudorandomness; added RSA ciphersuites; made key validation clearer; renamed hash\_to\_curve to encode\_to\_curve to be consistent with the hash\_to\_curve draft; allowed a more general salt in hashing, added the public key as input to ECVRF\_challenge\_generation, and added an explanation about the salt.

12 - Added k\_string to edwards25519 test vectors

13 - Clarified key validation for edwards25519 and addressed IRTF Chair comments

14 - Addressed IRSG review comments, which resulted in a substantial reworking of section 3.

15 - Added RSA-FDH-VRF test vectors.

## 9. Contributors

This document would not be possible without the work of Moni Naor, Sachin Vasant, and Asaf Ziv. Chloe Martindale provided a thorough cryptographer's review. Liliya Akhmetzyanova, Tony Arcieri, Gary Belvin, Mario Cao Cueto, Brian Chen, Sergey Gorbunov, Shumon Huque, Gorka Irazoqui Apecechea, Marek Jankowski, Burt Kaliski, Mallory Knodel, David C. Lawrence, Derek Ting-Haye Leung, Antonio Marcedone, Piotr Nojszewski, Chris Peikert, Colin Perkins, Trevor Perrin, Sam Scott, Stanislav Smyshlyaev, Adam Suhl, Nick Sullivan, Christopher Wood, Jiayu Xu, and Annie Yousar provided valuable input to this draft. Christopher Wood, Malte Thomsen, Marcus Rasmussen, and Tobias Vestergaard provided independent verification of the test

vectors. Riad Wahby helped this document align with draft-irtf-cfrg-hash-to-curve.

## 10. References

### 10.1. Normative References

- [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>>.
- [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>>.
- [RFC5114] Lepinski, M. and S. Kent, "Additional Diffie-Hellman Groups for Use with IETF Standards", RFC 5114, DOI 10.17487/RFC5114, January 2008, <<https://www.rfc-editor.org/info/rfc5114>>.
- [RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <<https://www.rfc-editor.org/info/rfc6234>>.
- [RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <<https://www.rfc-editor.org/info/rfc8032>>.
- [RFC6979] Pornin, T., "Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)", RFC 6979, DOI 10.17487/RFC6979, August 2013, <<https://www.rfc-editor.org/info/rfc6979>>.
- [I-D.irtf-cfrg-hash-to-curve] Faz-Hernandez, A., Scott, S., Sullivan, N., Wahby, R. S., and C. A. Wood, "Hashing to Elliptic Curves", Work in Progress, Internet-Draft, draft-irtf-cfrg-hash-to-curve-16, 15 June 2022, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hash-to-curve-16>>.
- [FIPS-186-4] National Institute for Standards and Technology, "Digital Signature Standard (DSS)", FIPS PUB 186-4, July 2013, <<https://csrc.nist.gov/publications/detail/fips/186/4/final>>.
- [SECG1] Standards for Efficient Cryptography Group (SECG), "SEC 1: Elliptic Curve Cryptography", Version 2.0, May 2009, <<http://www.secg.org/sec1-v2.pdf>>.

### 10.2. Informative References

- [ANSI.X9-62-2005] "Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)", ANSI X9.62, 2005.



- [BreHen19] Breitner, J. and N. Heninger, "Biased Nonce Sense: Lattice Attacks against Weak ECDSA Signatures in Cryptocurrencies", in Financial Cryptography, 2019, <<https://eprint.iacr.org/2019/023>>.
- [DGKR18] David, B., Gazi, P., Kiayias, A., and A. Russell, "Ouroboros Praos: An adaptively-secure, semi-synchronous proof-of-stake protocol", in Advances in Cryptology - EUROCRYPT, 2018, <<https://eprint.iacr.org/2017/573>>.
- [GHMVZ17] Gilad, Y., Hemo, R., Micali, Y., Vlachos, Y., and Y. Zeldovich, "Algorand: Scaling Byzantine Agreements for Cryptocurrencies", in Proceedings of the 26th Symposium on Operating Systems Principles (SOSP), 2017, <<https://eprint.iacr.org/2017/454>>.
- [GNPRVZ15] Goldberg, S., Naor, M., Papadopoulos, D., Reyzin, L., Vasant, S., and A. Ziv, "NSEC5: Provably Preventing DNSSEC Zone Enumeration", in NDSS, 2015, <<https://eprint.iacr.org/2014/582.pdf>>.
- [I-D.vcelak-nsec5] Vcelak, J., Goldberg, S., Papadopoulos, D., Huque, S., and D. C. Lawrence, "NSEC5, DNSSEC Authenticated Denial of Existence", Work in Progress, Internet-Draft, draft-vcelak-nsec5-08, 29 December 2018, <<https://datatracker.ietf.org/doc/html/draft-vcelak-nsec5-08>>.
- [MRV99] Micali, S., Rabin, M., and S. Vadhan, "Verifiable Random Functions", in FOCS, 1999, <<https://dash.harvard.edu/handle/1/5028196>>.
- [PWHVNRG17] Papadopoulos, D., Wessels, D., Huque, S., Vcelak, J., Naor, M., Reyzin, L., and S. Goldberg, "Making NSEC5 Practical for DNSSEC", in ePrint Cryptology Archive 2017/099, February 2017, <<https://eprint.iacr.org/2017/099>>.
- [X25519] Bernstein, D.J., "How do I validate Curve25519 public keys?", 2006, <<https://cr.yp.to/ecdh.html#validate>>.

## Appendix A. Test Vectors for the RSA-FDH-VRF ciphersuites

The test vectors in this section were generated using code at <https://github.com/reyzin/rsa-fdh-vrf>.

There are three keys used in the nine examples below. First, we provide the keys. They are shown in hexadecimal big-endian notation.

2048-bit key:

```
p =
efb52a568fa3038fff853e2183791c6bc81ceee86d20e8f9b6401dc79a8f1f62
48d3a25fdb3f99245fce41667da038f59745b87cc1aed8b4a9c1d74e7d5c16cf7
343f2b12f1b5055337369bf018fa07adc0d16f2164a516e80d2b4734f0c6563d6
ee6d4a9e1a54e300cfe9ee679afc3d14a152dfb49b6cfb208bbf921f764af
```



q =  
ecbca5ee88bbc635d8263aaba84f6502fdb2b4998a40f7c149133d840b6b1bd9a  
972fe2a981c770272b78fda213f76a062dd865dd116d4c8980975ee9347fe0f50  
0567e51d78dbee4a34e626051cf018d7feb72f19189525d4f70b6467d0cef5146  
33ab08a9e7a9ec632064b7b5e3e82128fe563757a614092fc5c624d10e1b  
n =  
ddaba77202bafb796b85bce98958aa58ae2d117cbc66a6e75c4c2af983985a30  
64eae93e2b03393256d94d75d6a6656b2956524ed8711898a0c3abae84371da0  
283bc5f433fc384d810a3c118ed302c0b03da16bee70b80ba3480e7acc1eb358b  
3f20f9e90cc4c8a7e2ba9e28b2a3800a5efbaa3c264f79b231f7cdc9577818df1  
bac60ef7a3f78a44f046fd29b0689556da7a7f61eefe67427f3f691aee0a4b1ef  
e2ee2e0e6091143ebb7d69254c9d8ab01ff5e0ad7329f566082f9251e64f436c5  
47e68de75351ea3a09746ceb7efed2d234121088aaed01696583c172ec88bc173  
a0d4d8ec43f4dcc18ff8379317e83ef9685536283368c9c6deb783075  
e = 010001  
d =  
d5c5ceab929a841e2a654536de4788f7f0a2a086d44bbb245f8aab3df00db924e  
8d644c3b502820f4cce98adacf09e73bc0e9762b50ae2b697aaa24914fa08b517  
58f59c07cf827341bb2a0597e126f9c69db031d60692c9cadf62842444696f082  
23154a1b0be752a325725748644e6d12935b1c66f983379773bcc8c65d06262e9  
3b5bb774dd2784265c23e9a7fc5e8871eb6bcc9968a6bc360a98874b623ec59f4  
1af0a9ecec6af095cb7e5aca11472363950dcbbf6f78fe003358b4fff0060a391  
daa45a1bd81c166b6221fb07e4f5da75e27d8d5fdbbf87ecbd7f5a4d804597070  
faaed22f197511b218788816689375245ddf7fa12337f3e7e898fb9d9

3072-bit key:

p =  
ee5adea28491084e6635bd73fd95649915a11da410d3f361c8eccc90a4b834251  
46da7b9e9d3994fd37d5fad7fb759ae451eb99b1102d4671ead23a2925133d19d  
f49cf9d7e9dcb69fd7555ca095338d0d2a84abb6825050eaf5fffaeff17ccb083  
3c6079081dfcbd98ced36a593557d29d64b0e0253ce2ee4e07fe2a06269dfe5ca  
230fad221a593a69d9534b2521c1b41d116cafdee02106228ff41433605453e23  
7777626953e79b46a84f50069e25b4f50496a928708abce30559eb183cf  
q =  
fb585bbc12f5695951f70a25e27682dc568acf56115ad749709b2a6e915cdd66d  
fa06db09b390c00b7c7ebee00845f73c999d8ea9352b1128bdf10113c7500b76  
a03f6b38d0920b5589961549be3d841ccc306f3edd600a53b4b9d4fa1249af87a  
f58dfb3ed694289477e853f7d062f58911f7bdb98033b001ee90f11b78f031cff  
ac2b5a07e11b01a2a6c1cda059a728f8253a5fff87267623253fc022d3993b27e2  
f344b99eb6072ff7c7ee160724f8fbc5a562be49247ffae42b55ea79dad5  
n =  
ea055cef495dec2d8fb3aef519ca87bd1575fa0ae15dd433f4a5f6c40d34ed6ba  
2388172ab7d2183ed970a669d427dc2774ced66a3f082b8e23e94e7de7532f4f3  
0bb4a5bbf2e1db2cba0752858a7c7a9bb892c5d6af7e90a7cee8f0097d14498c8  
b482f86348640af61b66640538e834f23ba8f906048db0e57b6fdc162ba2a8a0e  
aedd5423f23d8f89413223d89f473029cba11a211eb59e41fb8f0b8ddc651d115  
d9f07ac30296485a9adbd71cc5d9e4a448bd6d70785e838a978b2e66513eb897c  
962e85f00a36cc0a3a613183d8bd1572f895901eb8155af9797dbd4aa14726f41  
5712bf0eb29fa0a9e938cf5325def05d3af7e686227456d903233e316c8cc5034  
1615e59b665f0a4a2c32fc79469bdf89564481fb7afc27a7127741f79424e0  
a35cdc466dd33ef5a2067f75c86e06af9c03c68c6e78be5f1a4f49ea03569cd9f  
74c3a0ff290ca4ce2c2fa5b770ef8032b26a517c257b7b1c424622c5c04cf20f2  
290a268939e0cc79dfbac71842f94727b07bfafaded7db6c7f13b  
e = 010001  
d =  
6e68e957dbfd7c1862dc1b87780b9dcf0ff9016770bc9c09873b66194941d7621  
8bf2013c1e4df9326dd4402f5df110656d2ec8ea87a28b2a1cb74e590872aeb76

5fe772ea21c57d6ab4ba0fad019189273f05c061719afd14af02277dd28d67c5e  
f50b75b521ca51819b9bcb44cb7c82be66776a45f490050dc0171e77374f1ed00  
d06f8be09b711a9682107d8840d4a23edf6ac25441fdbf2b584dfa6a67cee21e  
b51c484f09416e11914e774713f1a17600fb9e4e99fbbd83fdcba4b09145dd980  
9449a1713777161c912d5d595362314b0ea9d1199e97780e8b3293a39af4019fc  
c746aaf78dbb7db06852c3358a9ed02ab1d15831a148b27b932c117445a4a6f51  
14edfa3ccc9a9862df714b78a5362aab5e30501b4a729af73e3cdcab19aac492  
8b668969780ad33d9df206d904b978a055f4abbc64987744526856e16ef559624  
53e3ed7a8055b0d79d051c50c94584ec7501dbd4856d7a21e43f25d8749e683cc  
a2f53f575af1d80f39d8e6932ffdf201d179cbf98314c4048c6c1

4096-bit key:

p =  
ac803464c8b2082153e15d5a0698d0a2990397fa01c1ed6171a5315e743c99fe  
b7acd31c37529d4f83405e657c390488d19f7da9ef9d9f9cfff4b460d2a26eb10f  
90cf4aaf55a19e21dc3bb697723a673e12bbc6580adc7bb72adaddf4682d656fff  
5b992e62379bc7b0ac977f2bfbcfac634e04ed597ef302684be72c6bf7db10b80  
f452d412d09e63e017acba378ccc6ea58e683e5641d1e72248f3201a5632f4af7  
525e91f9e0733731d264fe36802f416cb3e182b21e67a12e3bfba9a9cf40a45ff  
32addfae78063933120238ac61fbb995300a8602aa84f993bed375d6ccba86ad0  
c8efa5f0950aa2c92779febce9d05fa7a1f0d6e5c0d785de93c108297

q =  
feb39bb6ee78adfa524e9c0821f60c20d3cfff74f8b49731d67ea27d218bcb20c8  
7498d30dfd398bc23daff7b33dc330db93e6c0e5e6196e035446c6db7cfd9868  
b9518d94670b31f9c4d2109cf32c9cc8ac2fc4a6c2e1078510522c81610a81a70  
7997933ee24030b572a76ee51aa683312ecaa51d8558b3b19ccc6f5fc867354ae  
193fd5c4f5d5a7180c5ca1e90fcc42f6915dff69a3d1e49046f6c3ef841b262ba  
89ddcfdede2ed3caeb5bd594181a76f6f1ce01fc65c6f925f6d5b77037c2cbf7b60  
47e19f7b9c846c80238f1c8284c33bfd90c79de91381bb883b0de568aaf4b4a3c  
3f9c98f92e9f6a51f010bcc1dacfd72bfdfda29f527d7f4913153bef7

n =  
aba03a8d8527bfc0cbea1cb9a100f4ee7870aedd74a6406f108f7a07f37433602  
5357e256d655b342d73369102d03c7dcf3c14ed70aac7ebb62498c570068f71f1  
f165e14527f96d946ba839412252eacea604e7d6fd47a0bb9de776679fa9ad648  
5a076fda04a2015322626dcd2eb91d6b6248802e6d453eb4cbf5e1bfebed02d6a  
b36cfe3dd1e8b9749d4853a029940a0bed3aa3128fd8e2e6cd1115db15405bb38  
37012f56bdc5a6895ec5cc6bca52f7952cfe3c7d5d81d4d3d1c9a29a429eedfb  
f58da0a5b17480875b8071f49eb568fc8d8c023c83b3ed870c3775aaf0578485d  
757b4ab18d8e5fdb30c2b5586047e6203ab1636e376f1c7031f171e2807a2058e  
ce890cc8fae29ba819df76b45ddb514caee63db1c5e7a3af7468febff82bfe2eb  
79e3c5d1383b7ebef02e9cc1853f0f4486f7eb8fee23a2f794317ffd1c3947  
1086dfbfc0e3c0f412f917225f5c551557f38c11f172eca257e4b5908a571e4da  
a7c7434903701f21937df87d10de9b50ada97e65855d5e786db8f3f86248b55d9  
99ec31538bd1a409f3e13de46dccc05325774e89016708f8a96240ae1c16641e8  
b12ab07257e88aa50d3546e7a91073d85ed601775a3c08e9b7c242d20664dfd4e  
70a05218d9f2c7d760fab3cd772d9362527917cf5b51817e8c2aef51cb3b0dd8c  
b838097e513537f1d9c3c4708f44ed270db963c7d72cf11b1

e = 010001

d =  
1efd8dd524282b4deb04592f83cd226d353e53b5156d37d15652321ce16f281fc  
258487105b1f9a81054ef937bc89243bd7a01e56624d078d5a9021514c77a7b7e  
ceb230dd45fc9a36e4c1b9a4f347b9b29af3e3d14466fcb5242c398b389f70f9e  
7cf33ed54564e38c597720909e513ae8bb149060d1c6612e506e13d78e087c2cb  
b39e88c22cf73315c598dbd0ddf1276743ed04a943644c84949ef32d5e4702c80  
581e54a7fb18879be28b21008dc63182b45f2c190f1b748cd322efc39f2807c64  
b4d06023cb49583418e7b6ac0f447eb2abf48e2ad335583cbc8dff2760c2cce14  
62346326708336f7e374253ed213e990044927c52d29591f414571e509afc2396

```
a6af9843303a19673bcdec1e3fc7c0d6c3f43b4bf88ce83e2bdcfb5e39069fe32
800cf3f6f6d9917b8083a66ce23a9ab5b0c95bbcc6dfc21d38dadec20725b13c
e2954ba1bd45ec151a8877fed317cac60b2afaa96c826df6d1c48e7c10649dccc
75bdf905c362c6934da06c3ce30f5befc1cf776d7fda673625147b1108ecb5473
f7f588279533eb184d748230443694b9761b01532ba707563ffa4962321e44fdb
710025e8a6e00d29bf01ea040618ee111b5d79ac860083f91aa614777cc99d739
458f7c53d63cea7155b118068e0b30b35ed6d0cfc75672f18d075157a3ed31bfa
1ce2cea234357ec76117cc687c274636077abc437cb70a029
```

### A.1. RSA-FDH-VRF-SHA256

Example 1, using the 2048-bit key above:

```
alpha = (the empty string)
EM =
092ea69ca4f5630d4bd1012805ad23528a5f44c040829b4a0208491913ee39711
889bce5347765072efb0b7f8ad9798c830085d9babe10c29f1a649dbb9a64c93a
8cdaa325d37814faa15a1071ba81c39275f3cd66ce70fd21ee3acc7ac127c5de8
f2a816b05aff19e4e63451cfe51fef059b2547302387449b4df1ab8eaa5bfc84d
bbc5edf3b07eb8fe3fe2a93858bd0d55d6f0686f2eb449ed4c609b3083de04b49
d409a425509d89d282de806a6ce66892edc30337f780b15c7695b26383516f1fc
18f7eab52557c654467600e2e272ef41e7e4a060b42f7533bae603a7fa50f497a
64a1508b93826d99643a2001d1c958a7a06da0370668634d678a5de
pi =
14234ff8a9487e1b36a23086e258135b8a8a7ff2e23f19c0dfeca0c0a943f119e
bd336fdc292ef67b56e32ba06f9941893754a8b97c82f68974b2b34c17f6d43bf
d55eb110cd7ea3452d59a24e4ddb8d4cdf040c814e22e3537ca09c2e2dc5dd8ea
281e6492ad335378f9f437eed30c51eeee66ef14efb4000c75c802e9c5a6bb80
39c0258d4347981159d0ef6990b5e9c8ac2fb03915d7fff1ffa0626e2e11714a63
342e59124c1fcea8e2816c1d9a7751feaaa66cf6c82cd3c58ffde66460d98246a
b358cc33baefae4dfb0d191e9b6d6c0e3f92c35200408925dc8bef39b78d1259f
8163a5003a693555f05290ef2e68345f27c6e2a8847c5c919d92e7505
beta =
79f0615d4677fb72571889453644013f1a31b08d222e3cee349d64ce1c41045a
```

Example 2, using the 3072-bit key above:

```
alpha = 74657374 (4 bytes; ASCII "test")
EM =
20a059b7f7034d0d7696c63328cbbd4b40f7c656a632b4129915018fe6c5dee8b
5bde68ec2a5a78b1ca8483386e3a1a0fa07b4d329ea55facc3145c663ca90df5a
e46c903211a21bf908dc9a33bd09410cc09c7b4de5fbc79de3413bc80bccf2d3a
ca2fc9c60c776619849ed3e704057ac3d5deacff845d5bc8084ac730c19a14668
e53b5b8b90446b2272eaf59cdf985a7804c7b91cea1ce2582099b7b0f20163b11
d23110939dd62081b5aa46c62db76b2ac28473d2488970d480bdd8bef8cae9e81
274fe3f9925b012c1b55cba8c4291ec7433223cb872e422bb9e0d3775670d587e
403660ff440a9c11a18a488abc716ae36840b2ef5b0db4a90d88f91d79536cef3
78bf8e76d173288e26241df522a3cf6bece49c960e43a2d93e7bed10b90580c5b
3aff056507b4ef27368579832cb4aecc99c2d8ba402117457df5ae0ed28068ef
8b2e0d4582f8edacfcad02c83bfab778460b979e9e984827bbefe2b544c0f3ed7
15dde6dc1d7fc7c0f1f87d78aed8e148004b9f62e0321214c7c
pi =
69f6042d400dfad4bdb9974fb73d12ec7823c6632df6b0a97ebc14d8a443f74e1
eb1a99b37204ba5c7e53bdaf7e3e3fae9efe47cc01d0b061585c8d757ecf00663
b3e1bd447d55b6ebd066b814a8d9c4434b224e9cb053a1fd038a58f3bf6b0c75b
6f48f3c8d1ca398a730c133f86f244655f24c445324fdacd291d6d907f93efb24
b59e509f2f370392f5e262fc106292792352d93800f0a1e3a389786619a622f60
05cab78ea5f0b5b7ca91ad2a9c6c34fc4a3f9b0332b99e907ffa7f750cdc8342e
```

```
12da78f13ad49953bae1751c983ce3cd3335288ac856f85057a7f05acba6465a1
c6901ba30bc65b79fb7a847c42a5b4942d600ef316030f2ccafbc6f2e1ff0b46f
b5c8517cd98c93f81acf370cfdab559bb4270d07db5466e2342d56c476089f473
840434cbcbd1853b487a6df346208d12c17a48fe50b73b96f640a9761f570a51
6f6157432b83dd18a1d05cc27b6f283a02fcfda147cf1471772e469961004bde7
fa15857e7bf97b5a83c33fddb9f4b2e2488f4ed5f7463c93f30b
beta =
bfe966f3fabde6f38a2792ad59bc836bbca39de6eff64f15a42886deff6dfcc5
```

Example 3, using the 4096-bit key above:

```
alpha = 73616d706c65 (6 bytes; ASCII "sample")
EM =
17524fa1710b2f8a04e55da403b9b287b99a47afe9b81d3421482e3959b73b4d4
d4f4b52243ff2bfd2d29b1f030b521d0699065faa2b8903cca2b24cff42956123
4fcbdb7bcccdac61b7dcb7bc61cd857287b4b42357adbd2fc83ecfc0d5bc199e1f
6e298b5e470bfc540bc85e933b02035792d65d861096dc03f048cae51c9adc6c1
ec09e7f5e595681b3d3976d94ba1a65c83c7e82503db5478d3d91b2e00a0f24e7
ffee1faed68aa62ad7ba4b2912ceb636064766f0535d3ca1369760d8edebc3c8d
7f5b4de784b644b59e44e24e436298cc33a3cd0f676d6fa0b76ca3b9b11aa68e0
789e83bd27b3af08518b9a5eb5f34f4953a79dc25c1285b20fa73e558dd99638e
b51bf89c80d7989f6e925d8ca5ed1d3f29cc1e1065400e4abdbbcf898791be12c
5ae25661bf7de58a4cb6608c9a4dcc18150638068bb6452b25589ae0a943a67f0
24dd4b5d9e7940c01886f798316156e5771c19457f9104618e271ae7863b65fd0
7f87fcd7862690115ce2d963eeac60f78b47c037d6ed3000b43d8149cee08df10
a97158ee1daaf0a3963d23fb6ab0615891734e3039417d8ce03bfc18920c832a4
0385de95d99b546b4bd24ecbf2e75e9158ae1769bcff444990f54aa40e6a14e0
aca52df00062afbf81f6ce8193c53f8d26ac71324fc1db878379178abd695cf04
a0fae3432d1efffa73bba15b4e81fbaf598146a0c3edafc
pi =
745cc4b6cb75b925194374cdf91b498e8d687c5d9cae1eb5352446c554c2c43ac
4aa3e2db5cf5e366df635ce156a277ebdbe78c5598588c98257069253127e57c9
735b498f2939f14e1d019795cbd74cee2693acda2666624f174e8f666494aa126
41bce0677acd20e5552d2690117bddb38678a18acdc380bd9d93f3b10960f9be0
c141fc14f5f30da324ff14020cb5b8aed9fbca3fc44b4973d8e5527bd81f5ae5d
a67e5cc995abd1f7f9cdd3fa89b243fd4d5d5086ddb4eed77a2851fda1d4463f5
ee037a4015aa40c420c2e609d5d0da4ef4a1622131022bdd9c9dc26d177b39266
3ea42050ef485fe9d53a8d28d84b82a21101bed5b213c82b578ce7c9c6f7c1bf9
eca3c248ace9f8835f3850158749111ce1a3bdf5766add72a95a47c8866a4817c
42c5cbd85d7bef52afab567e564f6625be9e04be6f7da012af68e6623ce4f29c6
92ba0b5f7665bb435a2168bd3b88aae0c6168bb87ea6977f35bb5ad833d96dd14
d340f2a67b241b01fd8caf415842fd0a9dd5f4ccf4e70f15efdb85332e1df2bb1
86be15f7195176435e01bfd00592710023c3a88ac0eea7189b32296f865a31037
5111a5f11b74d0c74b98dfe4c41ccbe695ea801ba47f37b878c1ed0ffff8302705
b63c891209ea63defa892969e015a86d97945189444524e5fb660f2b9d1dce337
a12e0d003ea6262ca3194515cc3aa10b1a03ac9dd6995b54d
beta =
b663c5f90da1c12cd5d0e6d049679459e6f79f9fe16bc8b8e7e4d64d66500bd9
```

#### A.2. RSA-FDH-VRF-SHA384

Example 4, using the 2048-bit key above:

```
alpha = (the empty string)
EM =
1fa5c0079423d46edb63a833abb2e6ecfd5f39d1f2bd68fc666274d9e8ed8ea8a
13411126861167a4ba1d014d5ae213372de6bb4227b12e68e16e13ce108536acb
25f7219c49388f757219716fcb74eb0245b826c7e47ca793864885684b7673e2f
```

8579f26e78d63a940eacb23bf7619290cb5cd20859482c410fbd6d83a61f89408  
66f512be7ac041fc23c3ee71d918ec994f3efa62f4f1f44eaa29f5b37a1e93e24  
73d8677fcbec312838379a3e05899ce44227c0c428fbd7d4f2d0b46cfd7254e3  
967b220f8661f5dfbce7a3bf19364f522914478cead3eff0f0e02d166c251319b  
cf86701af1c48436f49ceac990f52940f7da6ac6f5fdafa5c55dc77

pi =

cffe6067bd9a1285dc1e8e543e8582c1250407cbfbc2d01c4ddbc0d4ecb5edeb  
721fb33147cf95f3084f7ce611f9877814770b14b8a671abc7ff085cf5cbe91e7  
2d17f076d62db478d4758412a4e4b77a5591dc32b764a501d27e34e56189ba734  
7a96f141ed1290f8ef7c4ce4009a9aba0715cbd0148721ea72bce00a22e594604  
21a21e4d121fc0b4eda62479d93724afae7556abe66326487be38cfb795ac1968  
c33a3890f2d8c0f7dfbe88bc76f16cbfd2b0f7ee8663abfd7b789caa5f6c77dd1  
ca991c9a9cc532f7550ad6184c8ece12ca4bea7e67f32405416a1f83245b09d06  
e7b4214157fb444be12a2eddc4381678f2b862fb240fcedd2da7ffcb3

beta =

dc37e83f8de0e990abada5096a05ca74754cfe7fe8e46b831e24100919415415d  
cd5a305f5fb8195713cebc78649c8d1

Example 5, using the 3072-bit key above:

alpha = 74657374 (4 bytes; ASCII "test")

EM =

fa2fd7c735c961b43b01c005faefc4e39505ede3914076d4dee40d52acf727de1  
782386ad6e9e07faf7666c8f45fde93b024d97c40651b957cfcccc42b8596a68a  
5495c02313ed9ecbb705ab0689c38b9e57af035189e377ad50b4704004c2a973d  
9f7554204b03e8b925a973d41a9c3432246eb2eab2f729f03d3a63c9c38c0cc2b  
aa440ed5e2d61644405e4b5c1acaac85d8de75a4de00419a478e6c44a97b3e898  
75c318400ce8d75b84c416ffd501ba78dd3203f21c6610fcaa4d8fc94f45e80dc  
65b7e48967199e7acdb18d82413b7018192a6fa2da5d6838adb8e6139f8d12abc  
ce7d5fd20cfa031c4971e563d4863d498591dc652a937db5e0bfd68535e3c9db9  
6118874287c2291a5d3b29aa142795e60f1ade2c8c4d627ee678b652f5fded61f  
9a60d2fa9cf5fb5e6a7fd63d81c91ea2269388f0a96fae77da0957695779385c7  
5748956972ab1cb5e19ad3cc6a357b9fffd368ca985dd9c0e53dd42aff5985f7a2  
34af96ad9e34e459a958b808e858f6d7be2e964c33cefad9660

pi =

22c9278e7171183cf6a3ce108f0400e308a9177c39a171f77777c106c966eb041  
824ce43fa56c5c77576646dd110e0b5d7f838bd5b1d1bf2c1feb1520397dd52d3  
cea6dbb49d786aa3bf3f5235e7692e583d290c7192102a6e0cb64f5229a326d4d  
00267fd75aae9687167ea0d3d450b2d63519ad605e64c77438728a190a129b116  
3939a5b7b0721b8d81efbf99a96944f63bf80ecc932fe40402d67c3e099a317cd  
1d13ac6947096308050ea6dad18fdb0958ae565d07d29e619673798f52b8d1dfd  
bf29b4641324ea6db5b9f35870acde7bf68e0829534d1c1f43ca9a16861efd82f  
b883e35d581f613d2dfbc89d01a84fdf081a3a850f2e865188cd995857222160c  
54780dc310a6ec100b9bac30f3af92e641360cad8dc255b56fa28e88ffcbe8eb  
e6ba8557e4ec44a7d0ebef882ade36db0d89be71ecaa2b35026c9d328d2384b54  
ae68de2ea70160ddde9aced5a8d896590fc185b408732cc04a249eff275015949  
02bf3af4a3743c4da50c5d62a74746007dedb8358ecfef78c75ab

beta =

5bdf742667ad10080f4ca573ec66f751e82e4077d0db1b281df421af68d39412e  
70362dc5101b4b46e1e453eea7e0989

Example 6, using the 4096-bit key above:

alpha = 73616d706c65 (6 bytes; ASCII "sample")

EM =

1b1d2f330ee20b9b1754f5e6ee4126cf03ea2c7f4e8c52d96111da7f995090424  
28ec2f2eafdf41716c04a9976a26df77b3d4cea8b10b216e7786fb49e923d984a  
2ee13ad82b95783b68fcf3444b65d1353619602ae06e392dc030be105d4cebc6f



f8a647b79115357833bd5312b9d3f0df1a307e782ff4db8de0eb16259d6bfff2f5  
7b3dd60a57693d607c42013cbcfc140a77d4a651492854afbacc377ed6729d1c  
be72999a62a96190fb630e5abc54d5cbe93254426df4e2315dbc777360ffb2401  
b3dedbed1acacf4b3a63b5ff8e5ab6c0f8ffd9e2a34fffd68a8a593c64de2660d  
cedaaab13cd42ebf5720d49f3120b01f45f29d1f465e995b148c9266aa97793a9  
da2f38831d00f95f9688b1c50b52a4cbcc14f8287db822381cddd609c9c17828  
6b1bc2f94d7ef4d5ceb1293dd7b0fac16d1b3a8b2a7fcc454e52efd2de5a79939  
7fd55a909641fa775463f4808b520c3ebe0f94e2765f8538d91a4f53bb746e7d5  
eaf55b3876503760f5c015f9e52bc54bdfc9632028db5e88b7dc0b1e9f1661d0a  
9b3574e46311de8ef6278c4c14f68375763e5df0d4cf221a4c3e84493ed0c3698  
4c172d87b513857af4b6c10174dea9db6464e2bab210aa492987f0255d2c5588b  
1c79769da03b62f691d5c4e5fac65505c317bf96b4f70e97c002aa0a032b02e48  
ee3aead5703bde3186ce138f29ba36219fd3558af417945

pi =

89d801e364fd48c3b8672e7d7abd8a2a1e5bd36bb1e38af5aaefa2f01cde686fa  
2e33f88fdcc8eb3babc1c66cbbf7dcddb614041813990787be5feabe86bbec37  
3d2cbf7c080caa0e37a339d5de1d1455de28f9bef76cd72500c669e9cab4599b5  
5dc155d9dd5810174c170f646d3b0b459347c17347c0281eecf5055cf887d6bd0  
a2c962c77d5ff9355a53cea64c34ea0888110ec4eb32da69022e293a8843d4c06  
c9d6e020c594335720467a8337c6a939fb2c5d710f7bdab48a52f4e7483dae062  
c1b9f66f7c9038ba9ceef3d61cb4cc004319c94a267a2425b5f042cd7f1a17922  
d6596a88a6fefaef41fc87742f2badee7d7613179589b4d02611ac8fd7895d926  
f484f79542cdf7f034dd536c9596da2f588ac9840f6bb05875bd17107e7458cc5  
ea368a7699fd60c35b54253a718c26cf518712be9d86213b2c6bdd0b7dd169f9  
240e77bfc44223675454f9c5596ad2e6e607ea65011a721ecbfa993172ae372ae  
8743779b33278d25e11ced77b14bc481fce60e4fc10a8a211d8b359906509d683  
0c653d91c1a86865219db43f62c70ac6780644d2bd73c5c256527a3eae faebaf1  
f220732417e17dbf598636616f70f2088969ac796a853dc8a5f270a1c505797e8  
3d1675e4f40b59c150ca06c49bb0967a2e0c7e74eff9e182d0f7bb6f54f68fe78  
8b89d2191c87bbf7f3927978449c2174baa581dc64a9c58ed

beta =

8ec4d150788513c85eea3490d1a1ee1b7a397602d3f9c8b467527f09fab5252e5  
39f82e8002825608295ebba19644dd

### A.3. RSA-FDH-VRF-SHA512

Example 7, using the 2048-bit key above:

alpha = (the empty string)

EM =

7b08a7fff4e5d8fd4978ac5a0ddf48537a2bb3f952dc00affb25d747b4085c2  
9c68dddaa87378db32396219ce784acebe70699286318f42794927f546de5d85b  
befd80a02c3aa714fc17090baa0d0f7fb504e1af0b79ea02d41dc0bf576b8f214  
72dd4c55f96bd64772d3ebd0347abe74b9fdf35b754d0405e42ceb0e290fdd91e  
f766a3e27ff59cd86572d15274f6fd49400ec4d126145f3cae200d67d5d108999  
61658ece7dcbf41f1cca63f8b50399955416a1f55e0af116fac2a9fd1f2dc0085  
e6ad6c1c4bc12d9308d9a030c3e2ea7f037d1c98beb23d43d67a97e5bf52382b8  
e890c5967ab42f2010cac985d3a52fe726045746d4ffef901127646

pi =

a280db108df5ad6ac1bed67efbc5c6fc6da0d301b9c0b41d26e379cd223c613c5  
9d52c987e4baaa6de4de2103284ddd56aa0b662dfe8faa8f6a503b83b7c81f481  
e23a08761d49a151ada1d9daa132138bbd6f80204c7fa87716b120df957224f92  
b32a3a0f96c3b209080c408618a92382ab5575f10a57c24ee0ffd01d6b822dc36  
b27600bf36aafadfa0a1e65aa6a0f2fc1a9dcd207d9bf5181a9ca69120e154108  
00a26efd3ce619349592eeff7b1851737bd033a83f88744ddd3d3e782efb6d243  
8ffda22ddcaa32c821c6730a05d5bdab88c354809d615884744ff10276496bee7  
0b62feb6ed07a3948823e9ee2a453dbcd4450192c9de0128adfc7e147

beta =  
808ca1f8f66a48118aacb011394bd4e5f0011c89ca913943d467b81cc5c43086e  
588abdde061c3ee30f4c15b2a6b51ad0ada42c0737fd7b2206fb43d35c8ed22

Example 8, using the 3072-bit key above:

alpha = 74657374 (4 bytes; ASCII "test")  
EM =  
803b6618f0ad47da2db309b1f57807a286500020c9e2b1427ebd9ffff1104e3aa8  
a69210441cd58344bd810c4900825c84b1e5e36825f1e397df54c4419f8525d9a  
09a49e7fedc18b8d906cbd9ea831c55f2aaa0461e19ddd6ec9d14dafa1fcf49b7  
7458a65427b7f060bc7425538e5d3af1813752cb452d0b098514110399734d1f5  
5870c65ea3e799e6d9024a9e2fb95883e580578811a8c7d34b18f8fabc6c05fb9  
697335fcf2cb1b7576ee7a39dcff129e1f142106c45f30a8ae62370f576d1d1d8  
c6307fccfe25cb431f348dea81b6b7e6307bbefda2a0b23036653a612226392a5  
73b7d62e28f9fecc7f4be0bf0a3049ce8ed276b34130faa943aeedf962b42a3f  
c6c881bbf9a62039e9c0850f1393a2a02c6848d06c3520e086541d8af99ea3ef9  
f9da2e3b2bd3172682a47e5965899bc576b66e29a0b8dcf06871202a1a4e7f2ff  
19bdd9eac2241129a73d7d01303b80372ac62a0d5b6bfd1d7119e561ace229cd5  
3d2c9963d6127b9ade16dce4b07d1cd89247ffc438811dc8b3c  
pi =  
1aa828e0a751074fed2fa776fd29336a84987c064eeebcd3a8129fb688b47eb71  
09987d01db0c3624ba7cc75e2f1ad60f5e204a250a329048bc34df34d41bf6ea6  
651774d249ff9fc29aeabdf524400527aa1c4100b1af86b2dcc2e7aecc77f386b  
80f29ccd807cb705b5057431832dafe56733a1e7bfcba1d052a26d1a8512f297b  
5abad5afd64fbcf21b57531a9b2c8217c0d9f1c875c196d998f61e8017f6b6ebe  
7317545ed390e18305bc96abb1514ec271963d02bed91ccf029d022189f84bac8  
cfa216da54e39919118348dfea6f4f6532b49da7820ee2a21f42b762e107722ad  
0abf62271e0640d6b1c4d1a39b94ebd74b4283de2d6550cbdb1f29cac51671e9c  
8fc0ea0fddb082a14a221e0531615f2bcfba0d70e99e4997cb00f81fcab2b9556  
63220234a5e90f29bd08e6fa50dd92770d9e514e0f9eb27aee634877bcea681ff  
d7da2b5be2f80c1dde1243b17ac726401cf961c5ce06640eb93352402c1ebc59c  
92188c511b375d63124846b46017fe36dc13fc2d34dbd80b312e0  
beta =  
9202b6715b7921c5eb35572ed9ebb85848d3345efadd665049ce889be46322586  
d4177864c9179468473518c6b6ac2e9c85ae5ee5fcd3c0d8e6d4d8f18be6238

Example 9, using the 4096-bit key above:

alpha = 73616d706c65 (6 bytes; ASCII "sample")  
EM =  
289607894786ccf223b1e758232f3402aa50cd48bca3d2bd64b2ea9b4a69dc917  
56a42b2c1feaaa777763b9b7c91888c580433ac85f5fbb360f129ecee739f69b5  
60657687d38f7d43c84f6605005c38f56c91310eff27bae49b14d8a36542d69b7  
0efca8637b845be0f029c085a7b6aad6ca0eb65ffdf8d55d9538d3b54044ebbc  
26e092b2f3ddaace7aa5b4b234ec848bfdc72a4ecdc10c66fb845cfc5ad39756e  
7f26007cea0ebf1e878636f4e39308fe7a317a9b7e90051536ac028bc1a2ec200  
a5dad0e3b74717bd9e7ea620919e315799e0fea7b0895fcbc0b95686b2495dc23  
e3cd56a16652b0df0dbd3ca8a6c96b13973a0c31a5541e211229da8a56e588a61  
6c721baab8e2d30313008c2374887f147598468b378bf8949ab1165b9348245d0  
a6a5f795918fce05f0d072f81f78c7224e7f1c4684877d714f231d5775c887590  
861212eae2d174761158ac7d653b18f0d4b71362c0eb8a67bed1a48a4e7dc739b  
2b44694514cae7f192d236afb1ab2409f24dfa94a2d705d0087860d844ba04564  
bb6733ca20089417d74eaed86d7e68ced681e9d88a9c3d7e6a33927592820cb9a  
38d4539332e509296489e54cd6b8495f100c36debe1f719578b15e8a99cc8febc  
3212e81478aeca616a5230ae84e7079f52aefb2ec2a97157fb5d60e1ddcf03b13  
4be2c93ffba41d5d068750adc8df07e5a264640f7e586bd

```
pi =
17d7635cac33b0b72ea1c0afb1f681d1a96c5073ed9f88ed8bb54eb428d7b2db4
ee3355eee512ddc7af50694b37fd389f990278e22095b2582c78c4ed6070b0c73
82b0308b6d546141a9b0d6ebb3af97abd93c16a5d34a2d805d8aa444fed2297d0
17571a693d221fda094d40500ab9b203d397a7543e72b26b06e561d49696e01de
ebfed58b46611dd5a346e227d7519f8ffd1dc76a172c9f7f355c3e7e5ee7773ed
ab00a22af5c39367f3779da68ce6da9f8a594f5f6149012501181653572fe5549
a9c2bf36148b3bdc94feaedd600727fe5c11b7dcbfd73002ae08061cb4b84ba47
f1bf8c5d46bc2acb7cb4964a6dca7eedc396e663a64121d93dade8b83cea09d76
653cca1a8d20d6b7323a890651dc575025ba1be02d08c5946f50cde438339b06e
8633198da0d467d2cac7d98ae62dd71353f6fb19aa9daac851d0ce237b21db93b
91e518d5c1ac36cdf874975deb7aab3942acc3980f221f33ad1254eb8ac3138e0
87d045c4746e0b7eedcaf2a1a173559783eba8691555c1b0e468f8efe6501679b
760038ed6fc9ce6aa5ae24b3f1178713793c8e5ee96035a2f0ee02e2d10ac0986
13358d3cff10f4dff3437f2a48252c5d6805288fbd7ee05356f80db12aaeabf66
38677abf5b8eb2376fb76861cf1b817d5a0b878dae6beac44f078f37d982d941a
77582a77784fabd632e28d664d9f705f31e24d1ca623dfac7
beta =
6026f6defaf534cc79ce7c1b0370fb53e4825d2d44f549f696e06d693c39e852e
21a5e3b6ff093618dd277b40678957e1b90e8e6ca742efed30dc309b3b242b8
```

## Appendix B. Test Vectors for the ECVRF ciphersuites

The test vectors in this section were generated using code at <https://github.com/reyzin/ecvrf>.

### B.1. ECVRF-P256-SHA256-TAI

The example secret keys and messages in Examples 10 and 11 are taken from Appendix A.2.5 of [RFC6979].

Example 10:

```
SK = x =
c9afa9d845ba75166b5c215767b1d6934e50c3db36e89b127b8a622b120f6721
PK =
0360fed4ba255a9d31c961eb74c6356d68c049b8923b61fa6ce669622e60f29fb
6
alpha = 73616d706c65 (6 bytes; ASCII "sample")
try_and_increment succeeded on ctr = 1
H =
0272a877532e9ac193aff4401234266f59900a4a9e3fc3cfc6a4b7e467a15d06d
4
k =
0d90591273453d2dc67312d39914e3a93e194ab47a58cd598886897076986f77
U = k*B =
02bb6a034f67643c6183c10f8b41dc4babf88bff154b674e377d90bde009c2167
2
V = k*H =
02893ebee7af9a0faa6da810da8a91f9d50e1dc071240c9706726820ff919e839
4
pi =
035b5c726e8c0e2c488a107c600578ee75cb702343c153cb1eb8dec77f4b5071b
4a53f0a46f018bc2c56e58d383f2305e0975972c26feea0eb122fe7893c15af37
6b33edf7de17c6ea056d4d82de6bc02f
beta =
a3ad7b0ef73d8fc6655053ea22f9bede8c743f08bbbed3d38821f0e16474b505e
```



Example 11:

```
SK = x =
c9afa9d845ba75166b5c215767b1d6934e50c3db36e89b127b8a622b120f6721
PK =
0360fed4ba255a9d31c961eb74c6356d68c049b8923b61fa6ce669622e60f29fb
6
alpha = 74657374 (4 bytes; ASCII "test")
try_and_increment succeeded on ctr = 3
H =
02173119b4fff5e6f8afed4868a29fe8920f1b54c2cf89cc7b301d0d473de6b97
4
k =
5852353a868bdce26938cde1826723e58bf8cb06dd2fed475213ea6f3b12e961
U = k*B =
022779a2cafcb65414c4a04a4b4d2adf4c50395f57995e89e6de823250d91bc48
e
V = k*H =
033b4a14731672e82339f03b45ff6b5b13dee7ada38c9bf1d6f8f61e2ce592111
9
pi =
034dac60aba508ba0c01aa9be80377ebd7562c4a52d74722e0abae7dc3080ddb5
6c19e067b15a8a8174905b13617804534214f935b94c2287f797e393eb0816969
d864f37625b443f30f1a5a33f2b3c854
beta =
a284f94ceec2ff4b3794629da7cbafa49121972671b466cab4ce170aa365f26d
```

The example secret key in Example 12 is taken from Appendix L.4.2 of [\[ANSI.X9-62-2005\]](#).

Example 12:

```
SK = x =
2ca1411a41b17b24cc8c3b089cfd033f1920202a6c0de8abb97df1498d50d2c8
PK =
03596375e6ce57e0f20294fc46bdfcfd19a39f8161b58695b3ec5b3d16427c274
d
alpha =
4578616d706c65207573696e67204543445341206b65792066726f6d204170706
56e646978204c2e342e32206f6620414e53492e58392d36322d32303035 (62
bytes; ASCII "Example using ECDSA key from Appendix L.4.2 of
ANSI.X9-62-2005")
try_and_increment succeeded on ctr = 1
H =
0258055c26c4b01d01c00fb57567955f7d39cd6f6e85fd37c58f696cc6b7aa761
d
k =
5689e2e08e1110b4dda293ac21667eac6db5de4a46a519c73d533f69be2f4da3
U = k*B =
020f465cd0ec74d2e23af0abde4c07e866ae4e5138bde5dd1196b8843f380db8
4
V = k*H =
036cb6f811428fc4904370b86c488f60c280fa5b496d2f34ff8772f60ed24b2d1
d
pi =
03d03398bf53aa23831d7d1b2937e005fb0062cbefa06796579f2a1fc7e7b8c66
7d091c00b0f5c3619d10ecea44363b5a599cadc5b2957e223fec62e81f7b4825f
c799a771a3d7334b9186bdbbee87316b1
```

```
beta =
90871e06da5caa39a3c61578ebb844de8635e27ac0b13e829997d0d95dd98c19
```

## B.2. ECVRF - P256 - SHA256 - SSWU

The example secret keys and messages in Examples 13 and 14 are taken from Appendix A.2.5 of [[RFC6979](#)].

Example 13:

```
SK = x =
c9afa9d845ba75166b5c215767b1d6934e50c3db36e89b127b8a622b120f6721
PK =
0360fed4ba255a9d31c961eb74c6356d68c049b8923b61fa6ce669622e60f29fb
6
alpha = 73616d706c65 (6 bytes; ASCII "sample")
In SSWU: uniform_bytes =
5024e98d6067dec313af09ff0cbe78218324a645c2a4b0aae2453f6fe91aa3bd9
471f7b4a5fbf128e4b53f0c59603f7e
In SSWU: u =
df565615a2372e8b31b8771f7503bafc144e48b05688b97958cc27ce29a8d810
In SSWU: x1 =
e7e39eb8a4c982426fcff629e55a3e13516cf62c02c369b1e750316f5e94eb
In SSWU: gx1 is a nonsquare
H =
02b31973e872d4a097e2cfae9f37af9f9d73428fde74ac537dda93b5f18dbc584
2
k =
e92820035a0a8afe132826c6312662b6ea733fc1a0d33737945016de54d02dd8
U = k*B =
031490f49d0355ffcdf66e40df788bee93861917ee713acff79be40d20cc91a30
a
V = k*H =
03701df0228138fa3d16612c0d720389326b3265151bc7ac696ea4d0591cd053e
3
pi =
0331d984ca8fece9cbb9a144c0d53df3c4c7a33080c1e02ddb1a96a365394c788
8782fffde7b842c38c20c08de6ec6c2e7027a97000f2c9fa4425d5c03e639fb48
fde58114d755985498d7eb234cf4aed9
beta =
21e66dc9747430f17ed9efeda054cf4a264b097b9e8956a1787526ed00dc664b
```

Example 14:

```
SK = x =
c9afa9d845ba75166b5c215767b1d6934e50c3db36e89b127b8a622b120f6721
PK =
0360fed4ba255a9d31c961eb74c6356d68c049b8923b61fa6ce669622e60f29fb
6
alpha = 74657374 (4 bytes; ASCII "test")
In SSWU: uniform_bytes =
910cc66d84a57985a1d15843dad83fd9138a109afb243b7fa5d64d766ec9ca389
4fdcf46eb21a3972eb452a4232fd3
In SSWU: u =
d8b0107f7e7aa36390240d834852f8703a6dc407019d6196bda5861b8fc00181
In SSWU: x1 =
ccc747fa7318b9486ce4044adbbecaa084c27be6eda88eb7b7f3d688fd0968c7
In SSWU: gx1 is a square
```

H =  
03ccc747fa7318b9486ce4044adbbecaa084c27be6eda88eb7b7f3d688fd0968c  
7  
k =  
febc3451ea7639fde2cf41ffd03f463124ecb3b5a79913db1ed069147c8a7dea  
U = k\*B =  
031200f9900e96f811d1247d353573f47e0d9da601fc992566234fc1a5b37749a  
e  
V = k\*H =  
02d3715dcfee136c7ae50e95ffca76f4ca6c29ddf92a39c31a0d48e75c6605cd  
1  
pi =  
03f814c0455d32dbc75ad3aea08c7e2db31748e12802db23640203aebf1fa8db2  
743aad348a3006dc1caad7da28687320740bf7dd78fe13c298867321ce3b36b79  
ec3093b7083ac5e4daf3465f9f43c627  
beta =  
8e7185d2b420e4f4681f44ce313a26d05613323837da09a69f00491a83ad25dd

The example secret key in Example 15 is taken from Appendix L.4.2 of [\[ANSI.X9-62-2005\]](#).

Example 15:

SK = x =  
2ca1411a41b17b24cc8c3b089cfd033f1920202a6c0de8abb97df1498d50d2c8  
PK =  
03596375e6ce57e0f20294fc46bdfcfd19a39f8161b58695b3ec5b3d16427c274  
d  
alpha =  
4578616d706c65207573696e67204543445341206b65792066726f6d204170706  
56e646978204c2e342e32206f6620414e53492e58392d36322d32303035 (62  
bytes; ASCII "Example using ECDSA key from Appendix L.4.2 of  
ANSI.X9-62-2005")  
In SSU: uniform\_bytes =  
9b81d55a242d3e8438d3bcfb1bee985a87fd144802c9268cf9adeeee160e6e9ff7  
65569797a0f701cb4316018de2e7dd4  
In SSU: u =  
e43c98c2ae06d13839fedb0303e5ee815896beda39be83fb11325b97976efdce  
In SSU: x1 =  
be9e195a50f175d3563aed8dc2d9f513a5536c1e9aee1757d86c08d32d582a86  
In SSU: gx1 is a nonsquare  
H =  
022dd5150e5a2a24c66feab2f68532be1486e28e07f1b9a055cf38ccc16f6595f  
f  
k =  
8e29221f33564f3f66f858ba2b0c14766e1057adbd422c3e7d0d99d5e142b613  
U = k\*B =  
03a8823ff9fd16bf879261c740b9c7792b77fee0830f21314117e441784667958  
d  
V = k\*H =  
02d48fbb45921c755b73b25be2f23379e3ce69294f6cee9279815f57f4b422659  
d  
pi =  
039f8d9cdc162c89be2871cbcb1435144739431db7fab437ab7bc4e2651a9e99d  
5488405a11a6c7fc8defddd9e1573a563b7333aab4effe73ae9803274174c6592  
69fd39b53e133dcd9e0d24f01288de9a  
beta =  
4fbadf33b42a5f42f23a6f89952d2e634a6e3810f15878b46ef1bb85a04fe95a

### B.3. ECVRF - EDWARDS25519 - SHA512 - TAI

The example secret keys and messages in Examples 16, 17, and 18 are taken from Section 7.1 of [\[RFC8032\]](#).

Example 16:

```
SK =
9d61b19deffd5a60ba844af492ec2cc44449c5697b326919703bac031cae7f60
PK =
d75a980182b10ab7d54bfed3c964073a0ee172f3daa62325af021a68f707511a
alpha = (the empty string)
x =
307c83864f2833cb427a2ef1c00a013cfdff2768d980c0a3a520f006904de94f
try_and_increment succeeded on ctr = 0
H =
91bbbed02a99461df1ad4c6564a5f5d829d0b90cfc7903e7a5797bd658abf3318
k_string =
7100f3d9eadb6dc4743b029736ff283f5be494128df128df2817106f345b8594b
6d6da2d6fb0b4c0257eb337675d96eab49cf39e66cc2c9547c2bf8b2a6afae4
k =
8a49edbd1492a8ee09766befe50a7d563051bf3406cbffc20a88def030730f0f
U = k*B =
aef27c725be964c6a9bf4c45ca8e35df258c1878b838f37d9975523f09034071
V = k*H =
5016572f71466c646c119443455d6cb9b952f07d060ec8286d678615d55f954f
pi =
8657106690b5526245a92b003bb079ccd1a92130477671f6fc01ad16f26f723f2
6f8a57ccaed74ee1b190bed1f479d9727d2d0f9b005a6e456a35d4fb0daab1268
a1b0db10836d9826a528ca76567805
beta =
90cf1df3b703cce59e2a35b925d411164068269d7b2d29f3301c03dd757876ff6
6b71dda49d2de59d03450451af026798e8f81cd2e333de5cdf4f3e140fdd8ae
```

Example 17:

```
SK =
4ccd089b28ff96da9db6c346ec114e0f5b8a319f35aba624da8cf6ed4fb8a6fb
PK =
3d4017c3e843895a92b70aa74d1b7ebc9c982ccf2ec4968cc0cd55f12af4660c
alpha = 72 (1 byte)
x =
68bd9ed75882d52815a97585caf4790a7f6c6b3b7f821c5e259a24b02e502e51
try_and_increment succeeded on ctr = 1
H =
5b659fc3d4e9263fd9a4ed1d022d75eaacc20df5e09f9ea937502396598dc551
k_string =
42589bbf0c485c3c91c1621bb4bfe04aed7be76ee48f9b00793b2342acb9c167c
ab856f9f9d4febc311330c20b0a8afd3743d05433e8be8d32522ecdc16cc5ce
k =
d8c3a66921444cb3427d5d989f9b315aa8ca3375e9ec4d52207711a1fdb44107
U = k*B =
1dcb0a4821a2c48bf53548228b7f170962988f6d12f5439f31987ef41f034ab3
V = k*H =
fd03c0bf498c752161bae4719105a074630a2aa5f200ff7b3995f7bfb1513423
pi =
f3141cd382dc42909d19ec5110469e4feae18300e94f304590abdced48aed5933
```

```
bf0864a62558b3ed7f2fea45c92a465301b3bbf5e3e54ddf2d935be3b67926da3
ef39226bbc355bdc9850112c8f4b02
beta =
eb4440665d3891d668e7e0fc587f1b4bd7fbfe99d0eb2211ccec90496310eb5
e33821bc613efb94db5e5b54c70a848a0bef4553a41befc57663b56373a5031
```

Example 18:

```
SK =
c5aa8df43f9f837bedb7442f31dcb7b166d38535076f094b85ce3a2e0b4458f7
PK =
fc51cd8e6218a1a38da47ed00230f0580816ed13ba3303ac5deb911548908025
alpha = af82 (2 bytes)
x =
909a8b755ed902849023a55b15c23d11ba4d7f4ec5c2f51b1325a181991ea95c
try_and_increment succeeded on ctr = 0
H =
bf4339376f5542811de615e3313d2b36f6f53c0acfebb482159711201192576a
k_string =
38b868c335ccda94a088428cbf3ec8bc7955bfafe1f3bd2aa2c59fc31a0febc5
9d0e1af3715773ce11b3bbdd7aba8e3505d4b9de6f7e4a96e67e0d6bb6d6c3a
k =
5ffdbc72135d936014e8ab708585fda379405542b07e3bd2c0bd48437fbac60a
U = k*B =
2bae73e15a64042fceb062abe7e432b2eca6744f3e8265bc38e009cd577ecd5
V = k*H =
88cba1cb0d4f9b649d9a86026b69de076724a93a65c349c988954f0961c5d506
pi =
9bc0f79119cc5604bf02d23b4caede71393cedfbb191434dd016d30177ccb809
6bb474e53895c362d8628ee9f9ea3c0e52c7a5c691b6c18c9979866568add7a2d
41b00b05081ed0f58ee5e31b3a970e
beta =
645427e5d00c62a23fb703732fa5d892940935942101e456ecca7bb217c61c452
118fec1219202a0edcf038bb6373241578be7217ba85a2687f7a0310b2df19f
```

#### B.4. ECVRF-EDWARDS25519-SHA512-ELL2

The example secret keys and messages in Examples 19, 20, and 21 are taken from Section 7.1 of [\[RFC8032\]](#).

Example 19:

```
SK =
9d61b19deffd5a60ba844af492ec2cc44449c5697b326919703bac031cae7f60
PK =
d75a980182b10ab7d54bfed3c964073a0ee172f3daa62325af021a68f707511a
alpha = (the empty string)
x =
307c83864f2833cb427a2ef1c00a013cfdff2768d980c0a3a520f006904de94f
In Elligator2: uniform_bytes =
d620782a206d9de584b74e23ae5ee1db5ca5298b3fc527c4867f049dee6dd419b
3674967bd614890f621c128d72269ae
In Elligator2: u =
30f037b9745a57a9a2b8a68da81f397c39d46dee9d047f86c427c53f8b29a55c
In Elligator2: gx1 =
8cb66318fb2cea01672d6c27a5ab662ae33220961607f69276080a56477b4a08
In Elligator2: gx1 is a square
```

```
H =
b8066ebbb706c72b64390324e4a3276f129569eab100c26b9f05011200c1bad9
k_string =
b5682049fee54fe2d519c9afff73bbfad724e69a82d5051496a42458f817bed7a
386f96b1a78e5736756192aeb1818a20efb336a205ffede351cfe88dab8d41c
k =
55cbb247af9b8372259a97b2cfe656d78868deb33b203d51b9961c364522400
U = k*B =
762f5c178b68f0cddcc1157918edf45ec334ac8e8286601a3256c3bbf858edd9
V = k*H =
4652eba1c4612e6fce762977a59420b451e12964adbe4fbecd58a7aeff5860af
pi =
7d9c633ffeee27349264cf5c667579fc583b4bda63ab71d001f89c10003ab46f1
4adf9a3cd8b8412d9038531e865c341cafa73589b023d14311c331a9ad15ff2fb
37831e00f0acaa6d73bc9997b06501
beta =
9d574bf9b8302ec0fc1e21c3ec5368269527b87b462ce36dab2d14ccf80c53ccc
f6758f058c5b1c856b116388152bbe509ee3b9ecfe63d93c3b4346c1fbc6c54
```

Example 20:

```
SK =
4ccd089b28ff96da9db6c346ec114e0f5b8a319f35aba624da8cf6ed4fb8a6fb
PK =
3d4017c3e843895a92b70aa74d1b7ebc9c982ccf2ec4968cc0cd55f12af4660c
alpha = 72 (1 byte)
x =
68bd9ed75882d52815a97585caf4790a7f6c6b3b7f821c5e259a24b02e502e51
In Elligator2: uniform_bytes =
04ae20a9ad2a2330fb33318e376a2448bd77bb99e81d126f47952b156590444a9
225b84128b66a2f15b41294fa2f2f6d
In Elligator2: u =
3092f033b16d4d5f74a3f7dc7091fe434b449065152b95476f121de899bb773d
In Elligator2: gx1 =
25d7fe7f82456e7078e99fdb24ef2582b4608357cdba9c39a8d535a3fd98464d
In Elligator2: gx1 is a nonsquare
H =
76ac3ccb86158a9104dff819b1ca293426d305fd76b39b13c9356d9b58c08e57
k_string =
88bf479281fd29a6cbdf6d7e2c5ec0024d92f14eaed58f43f22f37c4c37f1d41
e65c036fbf01f9fba11d554c07494d0c02e7e5c9d64be88ef78cab7544e444d
k =
9565956daedf376cad61b829b2a4d21ba1b52e9b3e2457477a64630a9711003
U = k*B =
8ec26e77b8cb3114dd2265fe1564a4efb40d109aa3312536d93dfe3d8d80a061
V = k*H =
fe799eb5770b4e3a5a27d22518bb631db183c8316bb552155f442c62a47d1c8b
pi =
47b327393ff2dd81336f8a2ef10339112401253b3c714eeda879f12c509072ef0
55b48372bb82efbdce8e10c8cb9a2f9d60e93908f93df1623ad78a86a028d6bc0
64dbfc75a6a57379ef855dc6733801
beta =
38561d6b77b71d30eb97a062168ae12b667ce5c28cacdf76bc88e093e4635987
cd96814ce55b4689b3dd2947f80e59aac7b7675f8083865b46c89b2ce9cc735
```

Example 21:

```
SK =
c5aa8df43f9f837bedb7442f31dcb7b166d38535076f094b85ce3a2e0b4458f7
PK =
fc51cd8e6218a1a38da47ed00230f0580816ed13ba3303ac5deb911548908025
alpha = af82 (2 bytes)
x =
909a8b755ed902849023a55b15c23d11ba4d7f4ec5c2f51b1325a181991ea95c
In Elligator2: uniform_bytes =
be0aed556e36cdfddf8f1eeddbb7356a24fad64cf95a922a098038f215588b216
beabbfe6acf20256188e883292b7a3a
In Elligator2: u =
f6675dc6d17fc790d4b3f1c6acf689a13d8b5815f23880092a925af94cd6fa24
In Elligator2: gx1 =
a63d48e3247c903e22fdfb88fd9295e396712a5fe576af335dbe16f99f0af26c
In Elligator2: gx1 is a square
H =
13d2a8b5ca32db7e98094a61f656a08c6c964344e058879a386a947a4e189ed1
k_string =
a7ddd74a3a7d165d511b02fa268710ddbb3b939282d276fa2efcfa5aaf79cf576
087299ca9234aacd7cd674d912deba00f4e291733ef189a51e36c861b3d683b
k =
1fda4077f737098b3f361c33a36cccafd7e9e9b720e1f84011254e25f37eed02
U = k*B =
a012f35433df219a88ab0f9481f4e0065d00422c3285f3d34a8b0202f20bac60
V = k*H =
fb613986d171b3e98319c7ca4dc44c5dd8314a6e5616c1a4f16ce72bd7a0c25a
pi =
926e895d308f5e328e7aa159c06eddb56d06846abf5d98c2512235eaa57fdce3
5b46edfc655bc828d44ad09d1150f31374e7ef73027e14760d42e77341fe05467
bb286cc2c9d7fde29120a0b2320d04
beta =
121b7f9b9aaaa29099fc04a94ba52784d44eac976dd1a3cca458733be5cd090a7
b5fbd148444f17f8daf1fb55cb04b1ae85a626e30a54b4b0f8abf4a43314a58
```

## Authors' Addresses

Sharon Goldberg  
Boston University  
111 Cummington Mall  
Boston, MA 02215  
United States of America

Email: [goldbe@cs.bu.edu](mailto:goldbe@cs.bu.edu)

Leonid Reyzin  
Boston University and Algorand  
111 Cummington Mall  
Boston, MA 02215  
United States of America

Email: [reyzin@bu.edu](mailto:reyzin@bu.edu)

Dimitrios Papadopoulos  
Hong Kong University of Science and Technology

Clearwater Bay  
Hong Kong

Email: [dipapado@cse.ust.hk](mailto:dipapado@cse.ust.hk)

Jan Vcelak  
NS1  
16 Beaver St  
New York, NY 10004  
United States of America

Email: [jvcelak@ns1.com](mailto:jvcelak@ns1.com)