CFRG S. Goldberg
Internet-Draft L. Reyzin
Intended status: Standards Track Boston University
Expires: March 18, 2019 D. Papadopoulos
Hong Kong University of Science and Techology
J. Vcelak
NS1
September 14, 2018

Verifiable Random Functions (VRFs)
draft-irtf-cfrg-vrf-03

Abstract

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

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

1.1. Rationale

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

A key application of the VRF is to provide privacy against offline enumeration (e.g. dictionary attacks) on data stored in a hash-based data structure. In this application, a Prover holds the VRF private 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 public VRF 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 strucuture.

1.2. 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 [RFC2119].

1.3. Terminology

The following terminology is used through this document:

SK: The private key for the VRF.

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 private VRF key SK and public VRF key PK.

Verifier: The Verifier holds the public VRF key PK.

2. VRF Algorithms

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

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

```
beta = VRF_hash(SK, alpha)
```

The VRF_hash algorithm is deterministic, in the sense that it always produces the same output beta given a pair of inputs (SK, alpha). The prover also uses the private key SK to construct a proof pi that beta is the correct hash output

```
pi = VRF_prove(SK, alpha)
```

The VRFs defined in this document allow anyone to deterministically obtain the VRF hash output beta directly from the proof value pi as

```
beta = VRF_proof_to_hash(pi)
```

Notice that this means that

```
VRF_hash(SK, alpha) = VRF_proof_to_hash(VRF_prove(SK, alpha))
```

and thus 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 VRF also comes with an algorithm

```
VRF_verify(PK, alpha, pi)
```

that outputs (VALID, beta = 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.

3.1. Full Uniqueness or Trusted Uniqueness

Uniqueness means that, for any fixed public VRF key and for any input alpha, there is a unique VRF output beta that can be proved to be valid. Uniqueness must hold even for an adversarial Prover that knows the VRF private key SK.

More precisely, "full uniqueness" states that a computationally-bounded adversary cannot choose a VRF public key PK, a VRF input alpha, and two proofs pi1 and pi2 such that VRF_verify(PK, alpha, pi1) outputs (VALID, beta1), VRF_verify(PK, alpha, pi2) outputs (VALID, beta2), and beta1 is not equal to beta2.

A slightly weaker security property called "trusted uniqueness" sufficies for many applications. Trusted uniqueness is the same as full uniqueness, but it must hold only if the VRF keys PK and SK were generated in a trustworthy manner. In other words, uniqueness might not hold if keys were generated in an invalid manner or with bad randomness.

3.2. Full Collison Resistance or Trusted Collision Resistance

Like any cryprographic hash function, VRFs need to be collision resistant. Collison resistance must hold even for an adversarial Prover that knows the VRF private key SK.

More precisely, "full collision resistance" states that it should be computationally infeasible for an adversary to find two distinct VRF inputs alpha1 and alpha2 that have the same VRF hash beta, even if that adversary knows the private VRF key SK.

For most applications, a slightly weaker security property called "trusted collision resistance" suffices. Trusted collision resistance is the same as collision resistance, but it holds only if PK and SK were generated in a trustworthy manner.

3.3. Full Pseudorandomness or Selective Pseudorandomness

Pseudorandomness ensures that when an adversarial Verifier 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 private VRF keys (PK, SK) were generated in a trustworthy manner. 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 computationally bounded adversary who does not know the private VRF key SK. This holds even if the adversary also gets to choose other VRF inputs alpha' and observe their corresponding VRF hash outputs beta' and proofs pi'.

With "full pseudorandomness", the adversary 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 which suffices in many applications. Here, the adversary must choose the target VRF input alpha independently of the public VRF key PK, and before it observes VRF outputs beta' and proofs pi' on inputs alpha' of its choice.

It is important to remember that the VRF output beta does not look random to the Prover, or to any other party that knows the private VRF key SK! Such a party can easily distinguish beta from a random value by comparing beta to the result of VRF_hash(SK, alpha).

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

Also, the VRF output beta may not look random if VRF key generation was not done in a trustworthy fashion. (For example, if VRF keys were generated with bad randomness.)

3.4. A random-oracle-like unpredictability property

Pseudorandomness, as defined in Section 3.3, does not hold if the VRF keys were generated adversarially. 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.

There is, however, a different type of unpredictability that is desirable in certain VRF applications (such as [GHMVZ17] and [KRDO17]). 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 uniform.

Although neither formal definitions nor proofs of this property have appeared in cryptographic literature, the VRF schemes presented in

this specification are believed to satisfy this property if the public key was generated in a trustworthy manner. Additionally, the ECVRF also satisifies this property even if the public key was not generated in a trustworthy manner, as long as the public key satisfies the key validation procedure in Section 5.6.

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

The RSA Full Domain Hash VRF (RSA-FDH-VRF) is a VRF that satisfies the "trusted uniqueness", "trusted collision resistance", and "full pseudorandomness" properties defined in Section 3. 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 document, the notation from [RFC8017] is used.

Parameters used:

(n, e) - RSA public key

K - RSA private key

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

Fixed options:

Hash - cryptographic hash function

hLen - output length in octets of hash function Hash

Primitives used:

I2OSP - Conversion of a nonnegative integer to an octet string as defined in Section 4.1 of [RFC8017]

OS2IP - Conversion of an octet string to a nonnegative integer as defined in Section 4.2 of [RFC8017]

RSASP1 - RSA signature primitive as defined in Section 5.2.1 of [RFC8017]

RSAVP1 - RSA verification primitive as defined in Section 5.2.2 of [RFC8017]

MGF1 - Mask Generation Function based on the hash function Hash as defined in Section B.2.1 of [RFC8017]

|| - octet string concatenation

4.1. RSA-FDH-VRF Proving

RSAFDHVRF_prove(K, alpha_string)

Input:

K - RSA private key

alpha_string - VRF hash input, an octet string

Output:

pi_string - proof, an octet string of length k

Steps:

1. one_string = 0x01 = I2OSP(1, 1), a single octet with value 1
2. EM = MGF1(one_string || I2OSP(k, 4) || I2OSP(n, k) || 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. two_string = 0x02 = I2OSP(2, 1), a single octet with value 2

2. beta_string = Hash(two_string || pi_string)

3. Output beta_string

4.3. RSA-FDH-VRF Verifying

RSAFDHVRF_verify((n, e), alpha_string, pi_string)

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 n

Output:

("VALID", beta_string), where beta_string is the VRF hash output, an octet string of length hLen; or
"INVALID"

Steps:

1. s = OS2IP(pi_string)

2. m = RSAVP1((n, e), s)

3. EM = I2OSP(m, k - 1)

4. one_string = 0x01 = I2OSP(1, 1), a single octet with value 1

5. EM' = MGF1(one_string || I2OSP(k, 4) || I2OSP(n, k) || alpha_string, k - 1)

6. If EM and EM' are equal, output ("VALID", RSAFDHVRF_proof_to_hash(pi_string)); else output "INVALID".

5. Elliptic Curve VRF (ECVRF)

The Elliptic Curve Verifiable Random Function (ECVRF) is a VRF that satisfies the trusted uniqueness, trusted collision resistance, and full pseudorandomness properties defined in Section 3. The security of this VRF follows from the decisional Diffie-Hellman (DDH) assumption in the random oracle model. Formal security proofs are in [PWHVNRG17].

To additionally satisfy "full uniqueness" and "full collision resistance", the Verifier MUST additionally perform the validation procedure specified in Section 5.6 upon receipt of the public VRF key.

Fixed options (specified in Section 5.5):

F - finite field

2n - length, in octets, of a field element in F, rounded up to the nearest even integer

E - elliptic curve (EC) defined over F

ptLen - length, in octets, of an EC point 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 (note that in the typical case, qLen equals 2n or is close to 2n)

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; must be at least 2n

suite_string - a single nonzero octet specifying the ECVRF ciphersuite, which determines the above options

Notation and primitives 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 - a raised to the power b

x*y - a multiplied by b

|| - octet string concatenation

ECVRF_hash_to_curve - collision resistant hash of strings to an EC point; options described in Section 5.4.1 and specified in Section 5.5.

ECVRF_nonce_generation - derives a pseudorandom nonce from SK and the input as part of ECVRF proving. Specified in Section 5.5

ECVRF_hash_points - collision resistant hash of EC points to an integer. Specified in Section 5.4.3.

Type conversions:

int_to_string(a, len) - conversion of nonnegative integer a to to octet string of length len as specified in Section 5.5.

string_to_int(a_string) - conversion of an octet string a_string to a nonnegative integer as specified in Section 5.5.

point_to_string - conversion of EC point to an ptLen-octet string as specified in Section 5.5

string_to_point - conversion of an ptLen-octet string to EC point as specified in Section 5.5. string_to_point returns INVALID if the octet string does not convert to a valid EC point.

arbitrary_string_to_point - conversion of an arbitrary octet string to an EC point as specified in Section 5.5

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. 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 private 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*B - VRF public key, an EC point

5.1. ECVRF Proving

Note: this function must have the VRF private key SK as input. Below we make it more efficient by supplying it also with the secret scalar x and the public key Y as additional inputs; however, each of these can be computed from SK if desired.

ECVRF_prove(Y, x, alpha_string)

Input:

SK - VRF private key

x - VRF secret scalar

Y = x*B - VRF public key

alpha_string = input alpha, an octet string

Output:

pi_string - VRF proof, octet string of length ptLen+n+qLen

Steps:

1. H = ECVRF_hash_to_curve(suite_string, Y, alpha_string)

2. h_string = point_to_string(H)

3. Gamma = x*H

4. k = ECVRF_nonce_generation(SK, h_string)

5. c = ECVRF_hash_points(H, Gamma, k*B, k*H)

6. s = (k + c*x) mod q

7. pi_string = point_to_string(Gamma) || int_to_string(c, n) || int_to_string(s, qLen)

8. 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+n+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)

2. If D is "INVALID", output "INVALID" and stop

3. (Gamma, c, s) = D

4. three_string = 0x03 = int_to_string(3, 1), a single octet with value 3

5. beta_string = Hash(suite_string || three_string || point_to_string(cofactor * Gamma))

6. Output beta_string

5.3. ECVRF Verifying

ECVRF_verify(Y, pi_string, alpha_string)

Input:

Y - public key, an EC point

pi_string - VRF proof, octet string of length ptLen+n+qLen

alpha_string - VRF input, octet string

Output:

("VALID", beta_string), where beta_string is the VRF hash output, octet string of length hLen; or
"INVALID"

Steps:

1. D = ECVRF_decode_proof(pi_string)

2. If D is "INVALID", output "INVALID" and stop

3. (Gamma, c, s) = D

4. H = ECVRF_hash_to_curve(suite_string, Y, alpha_string)

5. U = s*B - c*Y

6. V = s*H - c*Gamma

7. c' = ECVRF_hash_points(H, Gamma, U, V)

8. If c and c' are equal, output ("VALID", ECVRF_proof_to_hash(pi_string)); else output "INVALID"

5.4. ECVRF Auxiliary Functions

5.4.1. ECVRF Hash To Curve

The ECVRF_hash_to_curve algorithm takes in 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 in for proving and verfying. See Section 7.6 for further discussion.

The algorithms in this section are not compatible with each other; the choice of algorithm is made in Section 5.5.

5.4.1.1. ECVRF_hash_to_curve_try_and_increment

The following ECVRF_hash_to_curve_try_and_increment(suite_string, Y, alpha_string) algorithm implements ECVRF_hash_to_curve in a simple and generic way that works for any elliptic curve.

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_hash_to_try_and_increment(suite_string, Y, alpha_string)

Input:

suite_string - a single octet specifying ECVRF ciphersuite.

Y - public key, an EC point

alpha_string - value to be hashed, an octet string

Output:

H - hashed value, a finite EC point in G

Steps:

1. ctr = 0

2. PK_string = point_to_string(Y)

3. one_string = 0x01 = int_to_string(1, 1), a single octet with value 1

4. H = "INVALID"

5. While H is "INVALID" or H is EC point at infinity:

   A. ctr_string = int_to_string(ctr, 1)

   B. hash_string = Hash(suite_string || one_string || PK_string || alpha_string || ctr_string)

   C. H = arbitrary_string_to_point(hash_string)

   D. If H is not "INVALID" and cofactor > 1, set H = cofactor * H

   E. ctr = ctr + 1

6. Output H

5.4.1.2. ECVRF_hash_to_curve_elligator2_25519

The following ECVRF_hash_to_curve_elligator2_25519(suite_string, Y, alpha_string) algorithm implements ECVRF_hash_to_curve using the elligator2 algorithm from Section 5 of [BHKT13] (see also [I-D.irtf-cfrg-hash-to-curve]) exclusively for the Ed25519 elliptic curve (which the Edwards equivalent of Curve25519). It can be implemented with running time that is independent of the input alpha (so-called "constant-time").

ECVRF_hash_to_curve_elligator2_25519(suite_string, Y, alpha_string)

Input:

suite_string - a single octet specifying ECVRF ciphersuite.

alpha_string - value to be hashed, an octet string

Y - public key, an EC point

Output:

H - hashed value, a finite EC point in G

Fixed options:

p = 2^255-19, the size of the finite field F, a prime, for Curve25519

A = 486662, Montgomery curve constant for Curve25519

cofactor = 8 , the cofactor for Curve25519

Constraints on options:

output length of Hash is at least 16n (i.e., 256) bits

Steps:

1. PK_string = point_to_string(Y)

2. one_string = 0x01 = int_to_string(1, 1)
   (a single octet with value 1)

3. hash_string = Hash(suite_string || one_string || PK_string || alpha_string)

4. truncated_h_string = hash_string[0]...hash_string[31]

5. oneTwentySeven_string = 0x7F = int_to_string(127, 1)
   (a single octet with value 127)

6. truncated_h_string[31] = truncated_h_string[31] & oneTwentySeven_string
   (this step clears the high-order bit of octet 31)

7. r = string_to_int(truncated_h_string)

8. u = - A / (1 + 2*(r^2)) mod p
   (note: the inverse of (1+2*(r^2)) modulo p is guaranteed to exist)

9. w = u * (u^2 + A*u + 1) mod p
   (this step evaluates the Montgomery equation for Curve25519)

10. Let e equal the Legendre symbol of w and p
    (see note below on how to compute e)

11. If e is equal to 1 then final_u = u; else final_u = (-A - u) mod p
    (note: final_u is the Montgomery u-coordinate of the output; see note below on how to compute it)

12. y_coordinate = (final_u - 1) / (final_u + 1) mod p
    (note 1: y_coordinate is the Edwards coordinate corresponding to final_u)
    (note 2: the inverse of (final_u + 1) modulo p is guaranteed to exist)

13. h_string = int_to_string (y_coordinate, 32)

14. H_prelim = string_to_point(h_string)
    (note: string_to_point will not return INVALID by correctness of Elligator2)

15. Set H = cofactor * H_prelim

16. Output H

In order to make this algorithm run in time that is (almost) independent of the input alpha_string (so-called "constant-time"), implementers should pay particular attention to Steps 10 and 11 above. These steps can be implemented using the following approach:

```
e = w ^ ((p-1)/2) mod p

final_u = (e*u + (e-1) * (A/2)) mod p
```

The first step will produce a value e that is either 1 or p-1 (it is guaranteed not to be any other value, because w is guaranteed to be nonzero). Implementers should also ensure that the second step runs in the same amount of time regardless of e by ensuring that arithmetic in constant time.

Alternatively, let CMOV(result_if_1, result_if_0, selector) be the function that returns result_if_1 when selector is 1 and result_if_0 when selector is 0. If CMOV is implemented in constant time, then steps 12 and 13 above can be implemented as follows:

```
e = (w^((p-1)/2))+1 mod p

b = e/2

other_u = (-A-u) mod p

final_u = CMOV(u, other_u, b)
```

(Note that after the first step, e is either 0 or 2, and only the least significant byte of e is needed in the second step). CMOV can be implemented in constant time a variety of ways; for example, by expanding b from a single bit to an all-0 or all-1 string (accomplished by negating b in standard two's-complement arithmetic) and then applying bitwise XOR and AND operations as follows: other_x XOR ((x XOR other_x) AND b)

If having this algorithm run in constant time is not important, then there are much faster algorithms to compute the Legendre symbol (which is the same as the Jacobi symbol because p is a prime). See, for example, Section 12.3 of [ntb].

5.4.1.3. ECVRF_hash_to_curve_Simplified_SWU

The following ECVRF_hash_to_curve_Simplified_SWU(suite_string, Y, alpha_string) algorithm implements ECVRF_hash_to_curve using the simplified Shallue-Woestijne [SW06] and Ulas [Ulas07] algorithm from Section 7 of [BCIMRT10] (see also [I-D.irtf-cfrg-hash-to-curve]). It can be implemented with running time that is independent of the input alpha (so-called "constant-time"). Generally, this method can be used for any curve with prime p that is congruent to 3 modulo 4; however, the (very unlikely) case of d=0 in step 6 below may need to be handled differently depending on the curve equation, to ensure that the result is a point on the curve.

ECVRF_hash_to_curve_Simplified_SWU(suite_string, Y, alpha_string)

Input:

suite_string - a single octet specifying ECVRF ciphersuite.

alpha_string - value to be hashed, an octet string

Y - public key, an EC point

Output:

H - hashed value, a finite EC point in G

Fixed options:

a and b, constants for the Weierstrass form elliptic curve equation y^2 = x^3 + ax +b for the curve E

Steps:

1. PK_string = EC2OSP(Y)

2. one_string = 0x01 = I2OSP(1, 1), a single octet with value 1

3. h_string = Hash(suite_string || one_string || PK_string || alpha_string)

4. t = string_to_int(h_string) mod p

5. r = -(t^2) mod p

6. d = (r^2 + r) mod p
   (d is t^4-t^2 mod p)

7. If d = 0 then d_inverse = 0; else d_inverse = 1/d mod p
   (as long as Hash is secure, the case of d = 0 is an utterly improbably occurrence;
   the two cases can be combined into one by computing d_inverse = d^(p-2) mod p)

8. x = ((-b/a) * (1 + d_inverse)) mod p

9. w = (x^3 + a*x + b) mod p
   (this step evaluates the curve equation)

10. Let e equal the Legendre symbol of w and p
    (see note below on how to compute e)

11. If e is equal to 0 or 1 then final_x = x; else final_x = r * x mod p

(final_x is the x-coordinate of the output; see note below on how to compute it)

12. H_prelim = arbitrary_string_to_point(int_to_string(final_x, 2n))
(note: arbitrary_string_to_point will not return INVALID by correctness of Simple SWU)

13. If cofactor > 1, set H = cofactor * H; else set H = H_prelim

14. Output H

In order to make this algorithm run in time that is (almost) independent of the input (so-called "constant-time"), implementers should pay particular attention to Steps 10 and 11 above. These steps can be implemented using the following approach. Let CMOV(result_if_1, result_if_0, selector) be the function that returns result_if_1 when selector is 1 and result_if_0 when selector is 0. If arithmetic and CMOV are implemented in constant time, then steps 9 and 10 above can be implemented as follows:

```
e = (w ^ ((p-1)/2))+1 mod p
(At this point, e is 0, 1, or 2, as an integer.)

Let b = (e+1) / 2, where / denotes integer division with rounding
down.
(Note carefully that this step is integer, not modular, division.
Only the last byte of e is needed for this step.  This step
converts 0, 1, or 2 to 0 or 1.)

other_x = r * x mod p

final_x = CMOV(x, other_x, b)
```

CMOV can be implemented in constant time a variety of ways; for example, by expanding b from a single bit to an all-0 or all-1 string (accomplished by negating b in standard two's-complement arithmetic) and then applying bitwise XOR and AND operations as follows: other_x XOR ((x XOR other_x) AND b)

If having this algorithm run in constant time is not important, then there are much faster algorithms to compute the Legendre symbol (which is the same as the Jacobi symbol because p is a prime). See, for example, Section 12.3 of [ntb].

5.4.2. ECVRF Nonce Generation

The following subroutines generate the nonce value k in a deterministic pseudorandom fashion.

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 between 1 and q-1

The ECVRF_nonce_generation function is as specified in [RFC6979] Section 3.2 where

Step a is omitted

h_1 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 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

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].

ECVRF_nonce_generation_RFC8032(SK, h_string)

Input:

SK - an ECVRF secret key

h_string - an octet string

Output:

k - an integer between 0 and q-1

Steps:

1. hashed_sk_string = Hash (SK)

2. truncated_hashed_sk_string =
   hashed_sk_string[32]...hashed_sk_string[63]

3. k_string = Hash(truncated_hashed_sk_string || h_string)

4. k = string_to_int(k_string) mod q

5.4.3. ECVRF Hash Points

ECVRF_hash_points(P1, P2, ..., PM)

Input:

P1...PM - EC points in G

Output:

c - hash value, integer between 0 and 2^(8n)-1

Steps:

1. two_string = 0x02 = int_to_string(2, 1), a single octet with
   value 2

2. Initialize str = suite_string || two_string

3. for PJ in [P1, P2, ... PM]:
   str = str || point_to_string(PJ)

4. c_string = Hash(str)

5. truncated_c_string = c_string[0]...c_string[n-1]

6. c = string_to_int(truncated_c_string)

7. Output c

5.4.4. ECVRF Decode Proof

ECVRF_decode_proof(pi_string)

Input:

pi_string - VRF proof, octet string (ptLen+n+qLen octets)

Output:

"INVALID", or

Gamma - EC point

c - integer between 0 and 2^(8n)-1

s - integer between 0 and 2^(8qLen)-1

Steps:

1. let gamma_string = pi_string[0]...p_string[ptLen-1]

2. let c_string = pi_string[ptLen]...pi_string[ptLen+n-1]

3. let s_string =pi_string[ptLen+n]...pi_string[ptLen+n+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. Output Gamma, c, and s

5.5. ECVRF Ciphersuites

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

- suite_string = 0x01 = int_to_string(1, 1).

- 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, 2n = qLen = 32 and cofactor = 1.

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

- o The ECVRF_nonce_generation function is as specified in Section 5.4.2.1.

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

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

- o The point_to_string function converts an EC point to an octet string according to the encoding specified in Section 2.3.3 of [SECG1] with point compression on. This implies ptLen = 2n + 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.)

- o The string_to_point function converts an octet string to an EC point 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 an EC point.

- o arbitrary_string_to_point(h_string) = string_to_point(0x02 || h_string) (where 0x02 is a single octet with value 2, 0x02=int_to_string(2, 1)). The input h_string is a 32-octet string and the output is either an EC point or "INVALID".

- o The hash function Hash is SHA-256 as specified in [RFC6234], with hLen = 32.

- o The ECVRF_hash_to_curve function is as specified in Section 5.4.1.1.

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

- o This ciphersuite is identical to ECVRF-P256-SHA256-TAI except that the ECVRF_hash_to_curve function is as specified in Section 5.4.1.3 and suite_string = 0x02 = int_to_string(2, 1).

This document defines ECVRF-ED25519-SHA512-TAI as follows:

- o suite_string = 0x03 = int_to_string(3, 1).

- The EC group G is the Ed25519 elliptic curve with parameters defined in Table 1 of [RFC8032]. For this group, 2n = qLen = 32 and cofactor = 8.

- The private key and generation of the secret scalar and the public key are specified in Section 5.1.5 of [RFC8032]

- 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 an EC point to an octect string according to the encoding specified in Section 5.1.2 of [RFC8032]. This implies ptLen = 2n = 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 and implementation, the point_to_string function can be treated as the identity function.)

- The string_to_point function converts an octet string to an EC point 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 an EC point.

- arbitrary_string_to_point(h_string) = string_to_point(h_string[0]...h_string[31])

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

- The ECVRF_hash_to_curve function is as specified in Section 5.4.1.1.

This document defines ECVRF-ED25519-SHA512-Elligator2 as follows:

- This ciphersuite is identical to ECVRF-ED25519-SHA512-TAI except that the ECVRF_hash_to_curve function is as specified in Section 5.4.1.2 and suite_string = 0x04 = int_to_string(4, 1).

5.6. When the ECVRF Keys are Untrusted

The ECVRF as specified above is a VRF that satisfies the "trusted uniqueness", "trusted collision resistance", and "full pseudorandomness" properties defined in Section 3. In order to obtain "full uniqueness" and "full collision resistance" (which provide protection against a malicious VRF public key), the Verifier MUST perform the following additional validation procedure upon receipt of the public VRF key. The public VRF key MUST NOT be used if this procedure returns "INVALID".

Note that this procedure is not sufficient if the elliptic curve E or the point B, the generator of group G, is untrusted. If the prover is untrusted, the Verifier MUST obtain E and B from a trusted source, such as a ciphersuite specification, rather than from the prover.

This procedure supposes that the public key provided to the Verifier is an octet string. The procedure returns "INVALID" if the public key in invalid. Otherwise, it returns Y, the public key as an EC point.

5.6.1. ECVRF Validate Key

ECVRF_validate_key(PK_string)

Input:

PK_string - public key, an octet string

Output:

"INVALID", or

Y - public key, an EC point

Steps:

1. Y = string_to_point(PK_string)

2. If Y is "INVALID", output "INVALID" and stop

3. If cofactor*Y is the EC point at infinty, output "INVALID" and stop

4. Output Y

Note that if the cofactor = 1, then Step 3 need not multiply Y by the cofactor; instead, it suffices to output "INVALID" if Y is the point

at infinity. Moreover, when cofactor>1, it is not necessary to verify that Y is in the subgroup G; Step 3 suffices. Therefore, if the cofactor is small, the total number of points that could cause Step 3 to output "INVALID" may be small, and it may be more efficient to simply check Y against a fixed list of such points. For example, the following algorithm can be used for the Ed25519 curve:

1. Y = string_to_point(PK_string)

2. If Y is "INVALID", output "INVALID" and stop

3. y_string = PK_string

4. oneTwentySeven_string = 0x7F = int_to_string(127, 1)
   (a single octet with value 127)

5. y_string[31] = y_string[31] & oneTwentySeven_string
   (this step clears the high-order bit of octet 31)

6. bad_pk[0] = int_to_string(0, 32)

7. bad_pk[1] = int_to_string(1, 32)

8. bad_y2 = 2707385501144840649318225287225658788936804267575313519
   463743609750303402022

9. bad_pk[2] = int_to_string(bad_y2, 32)

10. bad_pk[3] = int_to_string(p-bad_y2, 32)

11. bad_pk[4] = int_to_string(p-1, 32)

12. bad_pk[5] = int_to_string(p, 32)

13. bad_pk[6] = int_to_string(p+1, 32)

14. If y_string is in bad_pk[0]...bad_pk[6], output "INVALID" and
    stop

15. Output Y

(bad_pk[0], bad_pk[2], bad_pk[3] each match two bad public keys, depending on the sign of the x-coordinate, which was cleared in step 5, in order to make sure that it does not affect the comparison. bad_pk[1] and bad_pk[4] each match one bad public key, because x-coordinate is 0 for these two public keys. bad_pk[5] and bad_pk[6] are simply bad_pk[0] and bad_pk[1] shifted by p, in case the y-coordinate had not been modular reduced by p. There is no need to

shift the other bad_pk values by p, because they will exceed 2^255. These bad keys, which represent all points of order 1, 2, 4, and 8, have been obtained by converting the points specified in [X25519] to Edwards coordinates.)

6. Implementation Status

A reference implementation of ECVRF-P256-SHA256-TAI, ECVRF-P256-SHA256-SWU, ECVRF-ED25519-SHA512-TAI, ECVRF-ED25519-SHA512-Elligator2 is available at <https://github.com/reyzin/ecvrf>. This implementation is neither secure nor especially effecient, but can be used to generate test vectors.

An implementation of the 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>. These implementations may be out of date as this spec has evolved.

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

An implementation by Yahoo! similar to the ECVRF is available at <https://github.com/r2ishiguro/vrf>.

An implementation similar to ECVRF 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 very similar to ECVRF-ED25519-SHA512-Elligator, called VXEdDSA, and specified here: <https://whispersystems.org/docs/specifications/xeddsa/>

7. Security Considerations

7.1. Key Generation

Applications that use the VRFs defined in this document MUST ensure that that the VRF key is generated correctly, using good randomness.

7.1.1. Uniqueness and collision resistance with untrusted keys

The ECVRF as specified in Section 5.1-Section 5.5 statisfies the "trusted uniqueness" and "trusted collision resistance" properties as long as the VRF keys are generated correctly, with good randomness. If the Verifier trusts the VRF keys are generated correctly, it MAY use the public key Y as is.

However, if the ECVRF uses keys that could be generated adversarially, then the the Verfier MUST first perform the validation procedure ECVRF_validate_key(PK) (specified in Section 5.6) upon receipt of the public key PK as an octet string. If the validation procedure outputs "INVALID", then the public key MUST not be used. Otherwise, the procedure will output a valid public key Y, and the ECVRF with public key Y satisfies the "full uniqueness" and "full collision resistance" properties.

The RSA-FDH-VRF statisfies the "trusted uniqueness" and "trusted collision resistance" properties as long as the VRF keys are generated correctly, with good randomness. These properties may not hold if the keys are generated adversarially (e.g., if RSA is not permutation). Meanwhile, the "full uniqueness" and "full collision resistance" are properties that hold even if VRF keys are generated by an adversary. The RSA-FDH-VRF defined in this document does not have these properties. However, if adversarial key generation is a concern, the RSA-FDH-VRF may be modifed to have these properties by adding additional cryptographic checks that its public key has the right form. These modifications are left for future specification.

7.1.2. Pseudorandomness with untrusted keys

Without good randomness, the "pseudorandomness" properties of the VRF 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.

7.2. Selective vs Full Pseudorandomness

[PWHVNRG17] presents cryptographic reductions to an underlying hard problem (e.g. Decisional Diffie Hellman for the ECVRF, or the standard RSA assumption for RSA-FDH-VRF) that prove the VRFs specificied in this document possess full pseudorandomness as well as selective pseudorandomness. However, the cryptographic reductions are tighter for selective pseudorandomness than for full pseudorandomness. This means the the VRFs have quantitavely stronger security guarentees for selective pseudorandomness.

Applications that are concerned about tightness of cryptographic reductions therefore have two options.

- o They may choose to ensure that selective pseudorandomness is sufficient for the application. That is, that pseudorandomness of outputs matters only for inputs that are chosen independently of the VRF key.

- o If full pseudorandomness is required for the application, the application 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. For both RSA-FDH-VRF and ECVRF the cryptographic strength of the hash function Hash may also potentially need to be increased.

7.3. Proper pseudorandom nonce for ECVRF

The security of the ECVRF defined in this document relies on the fact that nonce k used in the ECVRF_prove algorithm is chosen uniformly and pseudorandomly modulo q, and is unknown to the advesrary. Otherwise, an adversary may be able to recover the private VRF key x (and thus break pseudorandomness of the VRF) after observing several valid VRF proofs pi. The nonce generation methods specified in the ECVRF ciphersuites of Section 5.5 are designed with this requirement in mind.

7.4. Side-channel attacks

Side channel attacks on cryptographic primatives are an important issue. Here we discuss only one such side channel: timing attacks that can be used to leak information about the VRF input alpha. Implementers should take care to avoid side-channel attacks that leak information about the VRF private key SK (and the nonce k used in the ECVRF).

The ECVRF_hash_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.

Meanwhile, ECVRF-P256-SHA256-SWU and ECVRF-ED25519-SHA512-Elligator2 can be made to run in time constant in alpha.

7.5. Proofs Provide No Secrecy for 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.

7.6. 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 Initialized-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_hash_to_curve. The curve point H becomes the representative of alpha thereafter. Note that the suite_string octet and the public key are hashed together with alpha in ECVRF_hash_to_curve, which ensures that the curve (including the generator B) and the public key are included indirectly into subsequent hashes.

7.7. Hash function domain separation and future-proofing

Hashing is used for different purposes in the two VRFs (namely, in the RSA-FDH-VRF, in MGF1 and in proof_to_hash; in the ECVRF, in hash_to_curve, nonce_generation, hash_points, and proof_to_hash). The theoretical analysis assumes each of these functions is a separate random oracle. This analysis still holds even if the same hash function is used, as long as the four queries made to the hash function for a given SK and alpha are overwhelmingly unlikely to equal each other or to any queries made 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 first 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 to inputs given to hash_to_curve, proof_to_hash, and hash_points. 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

o the second octet of the input to the hash function used in hash_to_curve, proof_to_hash, and hash_points are all different

For the RSA VRF, if future designs need to specify variants of the design in this document, such variants should use different first octets in inputs to MGF1 and to the hash funciton used in proof_to_hash, in order to avoid the possibility that an adversary can obtain a VRF output under one variant, and then claim it was obtained under another variant

For the elliptic curve VRF, if future designs need to specify variants (e.g., additional ciphersuites) of the design 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. This way, the inputs to the hash_to_curve hash function used in producing H are guaranteed to be different; since all the other hashing done by the prover depends on H, inputs all the hash functions used by the prover will also be different as long as hash_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 determinsitic. 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 future-proofing. 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 an 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 paremeter; 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.

9. Contributors

This document also would not be possible without the work of Moni Naor (Weizmann Institute), Sachin Vasant (Cisco Systems), and Asaf Ziv (Facebook). Shumon Huque, David C. Lawerence, Trevor Perrin, Annie Yousar, Stanislav Smyshlyaev, Liliya Akhmetzyanova, Tony Arcieri, Sergey Gorbunov, Sam Scott, Nick Sullivan, Christopher Wood, Marek Jankowski, Derek Ting-Haye Leung, Adam Suhl, and Gary Belvin provided valuable input to this draft.

10. References

10.1. Normative References

[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>.

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

[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>.

[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>.

[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>.

[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>.

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

[BCIMRT10] Brier, E., Coron, J., Icart, T., Madore, D., Randriam, H., and M. Tibouchi, "Efficient Indifferentiable Hashing into Ordinary Elliptic Curves", in Advances in Cryptology - CRYPTO, 2010, <https://eprint.iacr.org/2009/340>.

[BHKT13] Bernstein, D., Hamburg, M., Krasnova, A., and T. Lange, "Elligator: elliptic-curve points indistinguishable from uniform random strings", in ACM SIGSAC Conference on Computer and Communications Security (CCS), 2013, <https://elligator.cr.yp.to/elligator-20130828.pdf>.

[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>.

[I-D.irtf-cfrg-hash-to-curve] Scott, S., Sullivan, N., and C. Wood, "Hashing to Elliptic Curves", draft-irtf-cfrg-hash-to-curve-01 (work in progress), July 2018.

[I-D.vcelak-nsec5] Vcelak, J., Goldberg, S., Papadopoulos, D., Huque, S., and D. Lawrence, "NSEC5, DNSSEC Authenticated Denial of Existence", draft-vcelak-nsec5-07 (work in progress), June 2018.

[KRDO17] Kiayias, A., Russell, A., David, B., and R. Oliynykov, "Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol", in Advances in Cryptology - CRYPTO, 2017, <https://eprint.iacr.org/2016/889>.

[MRV99] Michali, S., Rabin, M., and S. Vadhan, "Verifiable Random Functions", in FOCS, 1999, <https://dash.harvard.edu/handle/1/5028196>.

[ntb] Shoup, V., "A Computational Introduction to Number Theory and Algebra", 2008, <http://www.shoup.net/ntb/ntb-v2.pdf>.

[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.pdf>.

[SW06] Shallue, A. and C. van de Woestijne, "Construction of rational points on elliptic curves over finite fields", in Algorithmic Number Theory - ANTS, 2006, <https://works.bepress.com/andrew_shallue/1/>.

[Ulas07] Ulas, M., "Rational points on certain hyperelliptic curves over finite fields", in Bull. Polish Acad. Sci. Math., 2007, <https://arxiv.org/abs/0706.1448>.

[X25519] Bernstein, D., "How do I validate Curve25519 public keys?", 2006, <https://cr.yp.to/ecdh.html#validate>.

Appendix A. Test Vectors for the ECVRFs

The test vectors in this section were genereated using the reference implementation at <https://github.com/reyzin/ecvrf>.

A.1. ECVRF-P256-SHA256-TAI

These two example secret keys and messages are taken from Appendix A.2.5 of [RFC6979].

```
SK = x =
c9afa9d845ba75166b5c215767b1d6934e50c3db36e89b127b8a622b120f6721
PK =
0360fed4ba255a9d31c961eb74c6356d68c049b8923b61fa6ce669622e60f29fb6
alpha = 73616d706c65 (ASCII "sample")
try_and_increment succeded on ctr = 0
H =
02e2e1ab1b9f5a8a68fa4aad597e7493095648d3473b213bba120fe42d1a595f3e
k = c1aba586552242e6b324ab4b7b26f86239226f3cfa85b1c3b675cc061cf147dc
U = k*B =
02007fe22a3ed063db835a63a92cb1e487c4fea264c3f3700ae105f8f3d3fd391f
V = k*H =
03d0a63fa7a7fefcc590cb997b21bbd21dc01304102df183fb7115adf6bcbc2a74
pi = 029bdca4cc39e57d97e2f42f88bcf0ecb1120fb67eb408a856050dbfbcbf57c5
24193b7a850195ef3d5329018a8683114cb446c33fe16ebcc0bc775b043b5860dcb2e
553d91268281688438df9394103ab
beta =
59ca3801ad3e981a88e36880a3aee1df38a0472d5be52d6e39663ea0314e594c

SK = x =
c9afa9d845ba75166b5c215767b1d6934e50c3db36e89b127b8a622b120f6721
PK =
0360fed4ba255a9d31c961eb74c6356d68c049b8923b61fa6ce669622e60f29fb6
alpha = 74657374 (ASCII "test")
try_and_increment succeded on ctr = 0
H =
02ca565721155f9fd596f1c529c7af15dad671ab30c76713889e3d45b767ff6433
k = 7fc43fbc2aa51886139614792c613e672624b3fb8d0cf3fa6f52543d6a2fc26c
U = k*B =
037cd79cd8ab7b0324600b4d76c673a0726f3f1ff26b4b850d0c14b3aa272bf841
V = k*H =
02372cc8b56281e8d7f21ab7503c10af22164a4227e7433de0953a0df5e7a609bd
pi = 03873a1cce2ca197e466cc116bca7b1156fff599be67ea40b17256c4f34ba254
9c9c8b100049e76661dbcf6393e4d625597ed21d4de684e08dc6817b60938f3ff4148
823ea46a47fa8a4d43f5fa6f77dc8
beta =
dc85c20f95100626eddc90173ab58d5e4f837bb047fb2f72e9a408feae5bc6c1
```

This example secret key and message are taken from Appendix L.4.2 of [ANSI.X9-62-2005].

```
SK = x =
2ca1411a41b17b24cc8c3b089cfd033f1920202a6c0de8abb97df1498d50d2c8
PK =
03596375e6ce57e0f20294fc46bdfcfd19a39f8161b58695b3ec5b3d16427c274d
alpha = 4578616d706c65206f66204543445341207769746820616e7369703235367
23120616e64205348412d323536 (ASCII "Example of ECDSA with ansip256r1
and SHA-256")
try_and_increment succeded on ctr = 1
H =
02141e41d4d55802b0e3adaba114c81137d95fd3869b6b385d4487b1130126648d
k = 111e1505c8531c885dab6607a0962cd40a0af77637cdf183c7c9fb799dded43e
U = k*B =
02b3aceb619b90e811c8e50de73b27e65dd84669821055cb60dc1fa47199396c74
V = k*H =
02f117fe7daa8942d5492cc968784ced16025161b2dad374808eb7fbaf5eda5331
pi = 02abe3ce3b3aa2ab3c6855a7e729517ebfab6901c2fd228f6fa066f15ebc9b9d
41fd212750d9ff775527943049053a77252e9fa59e332a2e5d5db6d0be734076e98be
fcdefdcbaf817a5c13d4e45fbf9bc
beta =
e880bde34ac5263b2ce5c04626870be2cbff1edcdadabd7d4cb7cbc696467168
```

A.2. ECVRF-P256-SHA256-SWU

These two example secret keys and messages are taken from Appendix A.2.5 of [RFC6979].

```
SK = x =
c9afa9d845ba75166b5c215767b1d6934e50c3db36e89b127b8a622b120f6721
PK =
0360fed4ba255a9d31c961eb74c6356d68c049b8923b61fa6ce669622e60f29fb6
alpha = 73616d706c65 (ASCII "sample")
In SWU: t =
f1523667d029b9119a319a5bb316ff846691600e3552514ec4f93f9c84d65a4f
In SWU: w =
d8125c3ae82fc2b7f1c326b6f3dbfdf3583272336a60cb08efb84e002e98a3b3
In SWU: e = -1
H =
027827143876a58c2189402306c6ff6f7f9a7271067f3ed28eb63790d58a84fdd6
k = e15d8e7677f9473ae922d36977dbcc305a4ffd8149499dccfcb44fa097a2200c
U = k*B =
035dc0dee6903dba1a3e7cae4e2a960609e873e6e696cc8d5e56dfa8efccdfc97a
V = k*H =
03bec301c2930d69ed359eab2a54349d1431625c7a3ee0cfc2643ae5f8a21c3add
```

```
pi = 021d684d682e61dd76c794eef43988a2c61fbdb2af64fbb4f435cc2a842b0024
c35641fe838a72d0d9bc1bcf032f895f3b3f4c79d0f8f9d5705d83181fe82e19f4961
9eb8290930809b2b9651786e4f945
beta =
143f36bf7175053315693cfcfdff5aebb13e5eb9c47f897f53f81561993cfcd2

SK = x =
c9afa9d845ba75166b5c215767b1d6934e50c3db36e89b127b8a622b120f6721
PK =
0360fed4ba255a9d31c961eb74c6356d68c049b8923b61fa6ce669622e60f29fb6
alpha = 74657374 (ASCII "test")
In SWU: t =
e20da1d7386cb673deffec63d47ec65862dce55f113be168fa45cba2a6c1ddbc
In SWU: w =
0eed10be2937c902c9612d80b8ea5b0783f81c419faedd57efc84e6dfcfe2c72
In SWU: e = 1
H =
020e6c14efc8bc7150a3467aafa78be9856a2c6e405bdcc50f767fe638569d0172
k = 2addf2924eea6557e87acd635f08b54156cba70d718d8a3b6268af795cfdb7f2
U = k*B =
028347e823490cb50becb229cf059942afff39d6b276b987a384e45f29d1bf0dc3
V = k*H =
034759927f83caf0ed5d7ad1505844548051ef90a2f29de30efaf8eb811afb3342
pi = 0376b758f457d2cabdfaeb18700e46e64f073eb98c119dee4db6c5bb1eaf6778
06895ab451335f6adb792d40c68351929fce44068ffdcbbeac12f058b0365856ed5d8
6aadba1f54c9db13f9c8759589609
beta =
6b5bb622a6bc1387a7dcc4f46cfdcc3bce67669b32f3bc39e047c3b6cd3e65d9
```

This example secret key and message are taken from Appendix L.4.2 of [ANSI.X9-62-2005].

```
SK = x =
2ca1411a41b17b24cc8c3b089cfd033f1920202a6c0de8abb97df1498d50d2c8
PK =
03596375e6ce57e0f20294fc46bdfcfd19a39f8161b58695b3ec5b3d16427c274d
alpha = 4578616d706c65206f66204543445341207769746820616e7369703235367
23120616e64205348412d323536 (ASCII "Example of ECDSA with ansip256r1
and SHA-256")
In SWU: t =
e93da6ba2bca714061dc94c8c513343ad11bfc9678339e4a8bd86a08232aa6d7
In SWU: w =
76f564cca31934c80dd2a285ba43543df63a078b132c8f34d2ab1b7089cb3401
In SWU: e = -1
H =
02429690b91e1783cd0d7e393db07cc44b48c226cb837adb2282251cabf431a484
k = 2ba4cd5e9f7be946659555b3d052af618d2d3a14f5a756c8d56fab058d1831ff
```

```
U = k*B =
025943c217f48354e297156ac7dce8d2b50f63867acc23ba1aea87a66578392ca8
V = k*H =
0399ccafce764acf0db264183d97c1ae968daf661b9931a145bb8cbfc5f3f8f9a5
pi = 035e844533a7c5109ab3dffd04f2ef0d38d679101124f15243199ce92f0f2947
7cd29f8754f3bbdea3dd129560e9ba0c73ae7894a8d0c0e1ac01e5c2685da67009d96
e6ccdb634c7e0c5f38fa3e4908c02
beta =
be1dcb17e9815ac6acf819e7ad4b75e575eafad25915c2608959d780364fc912
```

A.3. ECVRF-ED25519-SHA512-TAI

These three example secret keys and messages are taken from Section 7.1 of [RFC8032].

```
SK = 9d61b19deffd5a60ba844af492ec2cc44449c5697b326919703bac031cae7f60
PK = d75a980182b10ab7d54bfed3c964073a0ee172f3daa62325af021a68f707511a
alpha = (the empty string)
x = 307c83864f2833cb427a2ef1c00a013cfdff2768d980c0a3a520f006904de94f
try_and_increment succeded on ctr = 0
H = 5b2c80db3ce2d79cc85b1bfb269f02f915c5f0e222036dc82123f640205d0d24
k = 647ac2b3ca3f6a77e4c4f4f79c6c4c8ce1f421a9baaa294b0adf0244915130f70
67640acb6fd9e7e84f8bc30d4e03a95e410b82f96a5ada97080e0f187758d38
U = k*B =
a21c342b8704853ad10928e3db3e58ede289c798e3cdfd485fbbb8c1b620604f
V = k*H =
426fe41752f0b27439eb3d0c342cb645174a720cae2d4e9bb37de034eefe27ad
pi = 9275df67a68c8745c0ff97b48201ee6db447f7c93b23ae24cdc2400f52fdb08a
1a6ac7ec71bf9c9c76e96ee4675ebff60625af28718501047bfd87b810c2d2139b73c
23bd69de66360953a642c2a330a
beta = a64c292ec45f6b252828aff9a02a0fe88d2fcc7f5fc61bb328f03f4c6c0657
a9d26efb23b87647ff54f71cd51a6fa4c4e31661d8f72b41ff00ac4d2eec2ea7b3

SK = 4ccd089b28ff96da9db6c346ec114e0f5b8a319f35aba624da8cf6ed4fb8a6fb
PK = 3d4017c3e843895a92b70aa74d1b7ebc9c982ccf2ec4968cc0cd55f12af4660c
alpha = 72 (1 byte)
x = 68bd9ed75882d52815a97585caf4790a7f6c6b3b7f821c5e259a24b02e502e51
try_and_increment succeded on ctr = 4
H = 08e18a34f3923db32e80834fb8ced4e878037cd0459c63ddd66e5004258cf76c
k = 627237308294a8b344a09ad893997c630153ee514cd292eddd577a9068e2a6f24
cbee0038beb0b1ee5df8be08215e9fc74608e6f9358b0e8d6383b1742a70628
U = k*B =
18b5e500cb34690ced061a0d6995e2722623c105221eb91b08d90bf0491cf979
V = k*H =
87e1f47346c86dbbd2c03eafc7271caa1f5307000a36d1f71e26400955f1f627
pi = 84a63e74eca8fdd64e9972dcda1c6f33d03ce3cd4d333fd6cc789db12b5a7b9d
03f1cb6b2bf7cd81a2a20bacf6e1c04e59f2fa16d9119c73a45a97194b504fb9a5c8c
f37f6da85e03368d6882e511008
```

```
beta = cddaa399bb9c56d3be15792e43a6742fb72b1d248a7f24fd5cc585b232c26c
934711393b4d97284b2bcca588775b72dc0b0f4b5a195bc41f8d2b80b6981c784e

SK = c5aa8df43f9f837bedb7442f31dcb7b166d38535076f094b85ce3a2e0b4458f7
PK = fc51cd8e6218a1a38da47ed00230f0580816ed13ba3303ac5deb911548908025
alpha = af82 (2 bytes)
x = 909a8b755ed902849023a55b15c23d11ba4d7f4ec5c2f51b1325a181991ea95c
try_and_increment succeded on ctr = 0
H = e4581824b70badf0e57af789dd8cf85513d4b9814566de0e3f738439becfba33
k = a950f736af2e3ae2dbcb76795f9cbd57c671eee64ab17069f945509cd6c4a7485
2fe1bbc331e1bd573038ec703ca28601d861ad1e9684ec89d57bc22986acb0e
U = k*B =
5114dc4e741b7c4a28844bc585350240a51348a05f337b5fd75046d2c2423f7a
V = k*H =
a6d5780c472dea1ace78795208aaa05473e501ed4f53da57e1fb13b7e80d7f59
pi = aca8ade9b7f03e2b149637629f95654c94fc9053c225ec21e5838f193af2b727
b84ad849b0039ad38b41513fe5a66cdd2367737a84b488d62486bd2fb110b4801a46b
fca770af98e059158ac563b690f
beta = d938b2012f2551b0e13a49568612effcbdca2aed5d1d3a13f47e180e012189
16e049837bd246f66d5058e56d3413dbbbad964f5e9f160a81c9a1355dcd99b453
```

A.4. ECVRF-ED25519-SHA512-Elligator2

These three example secret keys and messages are taken from Section 7.1 of [RFC8032].

```
SK = 9d61b19deffd5a60ba844af492ec2cc44449c5697b326919703bac031cae7f60
PK = d75a980182b10ab7d54bfed3c964073a0ee172f3daa62325af021a68f707511a
alpha = (the empty string)
x = 307c83864f2833cb427a2ef1c00a013cfdff2768d980c0a3a520f006904de94f
In Elligator: r =
9ddd071cd5837e591a3a40c57a46701bb7f49b1b53c670d490c2766a08fa6e3d
In Elligator: w =
c7b5d6239e52a473a2b57a92825e0e5de4656e349bb198de5afd6a76e5a07066
In Elligator: e = -1
H = 1c5672d919cc0a800970cd7e05cb36ed27ed354c33519948e5a9eaf89aee12b7
k = 868b56b8b3faf5fc7e276ff0a65aaa896aa927294d768d0966277d94599b7afe4
a6330770da5fdc2875121e0cbecbffbd4ea5e491eb35be53fa7511d9f5a61f2
U = k*B =
c4743a22340131a2323174bfc397a6585cbe0cc521bfad09f34b11dd4bcf5936
V = k*H =
e309cf5272f0af2f54d9dc4a6bad6998a9d097264e17ae6fce2b25dcbdd10e8b
pi = b6b4699f87d56126c9117a7da55bd0085246f4c56dbc95d20172612e9d38e8d7
ca65e573a126ed88d4e30a46f80a666854d675cf3ba81de0de043c3774f061560f55e
dc256a787afe701677c0f602900
beta = 5b49b554d05c0cd5a5325376b3387de59d924fd1e13ded44648ab33c21349a
603f25b84ec5ed887995b33da5e3bfcb87cd2f64521c4c62cf825cffabbe5d31cc
```

```
SK = 4ccd089b28ff96da9db6c346ec114e0f5b8a319f35aba624da8cf6ed4fb8a6fb
PK = 3d4017c3e843895a92b70aa74d1b7ebc9c982ccf2ec4968cc0cd55f12af4660c
alpha = 72 (1 byte)
x = 68bd9ed75882d52815a97585caf4790a7f6c6b3b7f821c5e259a24b02e502e51
In Elligator: r =
92181bd612695e464049590eb1f9746750d6057441789c9759af8308ac77fd4a
In Elligator: w =
7ff6d8b773bfbae57b2ab9d49f9d3cb7d9af40a03d3ed3c6beaaf2d486b1fe6e
In Elligator: e = 1
H = 86725262c971bf064168bca2a87f593d425a49835bd52beb9f52ea59352d80fa
k = fd919e9d43c61203c4cd948cdaea0ad4488060db105d25b8fb4a5da2bd40e4b83
30ca44a0538cc275ac7d568686660ccfd6323c805b917e91e28a4ab352b9575
U = k*B =
04b1ba4d8129f0d4cec522b0fd0dff84283401df791dcc9b93a219c51cf27324
V = k*H =
ca8a97ce1947d2a0aaa280f03153388fa7aa754eedfca2b4a7ad405707599ba5
pi = ae5b66bdf04b4c010bfe32b2fc126ead2107b697634f6f7337b9bff8785ee111
200095ece87dde4dbe87343f6df3b107d91798c8a7eb1245d3bb9c5aafb093358c13e
6ae1111a55717e895fd15f99f07
beta = 94f4487e1b2fec954309ef1289ecb2e15043a2461ecc7b2ae7d4470607ef82
eb1cfa97d84991fe4a7bfdfd715606bc27e2967a6c557cfb5875879b671740b7d8

SK = c5aa8df43f9f837bedb7442f31dcb7b166d38535076f094b85ce3a2e0b4458f7
PK = fc51cd8e6218a1a38da47ed00230f0580816ed13ba3303ac5deb911548908025
alpha = af82 (2 bytes)
x = 909a8b755ed902849023a55b15c23d11ba4d7f4ec5c2f51b1325a181991ea95c
In Elligator: r =
dcd7cda88d6798599e07216de5a48a27dcd1cde197ab39ccaf6a906ae6b25c7f
In Elligator: w =
2ceaa2c2ff3028c34f9fbe076ff99520b925f18d652285b4daad5ccc467e523b
In Elligator: e = -1
H = 9d8663faeb6ab14a239bfc652648b34f783c2e99f758c0e1b6f4f863f9419b56
k = 8f675784cdc984effc459e1054f8d386050ec400dc09d08d2372c6fe0850eaaa5
0defd02d965b79930dcbca5ba9222a3d99510411894e63f66bbd5d13d25db4b
U = k*B =
d6f8a95a4ce86812e3e50febd9d48196b3bc5d1d9fa7b6dfa33072641b45d029
V = k*H =
f77cd4ce0b49b386e80c3ce404185f93bb07463600dc14c31b0a09beaff4d592
pi = dfa2cba34b611cc8c833a6ea83b8eb1bb5e2ef2dd1b0c481bc42ff36ae7847f6
ab52b976cfd5def172fa412defde270c8b8bdfbaae1c7ece17d9833b1bcf31064fff7
8ef493f820055b561ece45e1009
beta = 2031837f582cd17a9af9e0c7ef5a6540e3453ed894b62c293686ca3c1e319d
de9d0aa489a4b59a9594fc2328bc3deff3c8a0929a369a72b1180a596e016b5ded
```

Authors' Addresses

Sharon Goldberg
Boston University
111 Cummington St, MCS135
Boston, MA 02215
USA

EMail: goldbe@cs.bu.edu

Leonid Reyzin
Boston University
111 Cummington St, MCS135
Boston, MA 02215
USA

EMail: reyzin@bu.edu

Dimitrios Papadopoulos
Hong Kong University of Science and Techology
Clearwater Bay
Hong Kong

EMail: dipapado@cse.ust.hkbu.edu

Jan Vcelak
NS1
16 Beaver St
New York, NY 10004
USA

EMail: jvcelak@ns1.com