

CFRG
Internet-Draft
Intended status: Standards Track
Expires: 10 August 2022

S. Goldberg
Boston University
L. Reyzin
Boston University and Algorand
D. Papadopoulos
Hong Kong University of Science and Technology
J. Vcelak
NS1
6 February 2022

Verifiable Random Functions (VRFs)
draft-irtf-cfrg-vrf-11

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 the 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 based on RSA and Elliptic Curves that are secure in the cryptographic random oracle model.

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 August 2022.

Copyright Notice

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

Internet-Draft

VRF

February 2022

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	3
1.1.	Rationale	3
1.2.	Requirements	3
1.3.	Terminology	4
2.	VRF Algorithms	4
3.	VRF Security Properties	5
3.1.	Full Uniqueness or Trusted Uniqueness	5
3.2.	Full Collision Resistance or Trusted Collision Resistance	5
3.3.	Full Pseudorandomness or Selective Pseudorandomness	6
3.4.	Some VRFs: Unpredictability Under Malicious Key Generation	7
4.	RSA Full Domain Hash VRF (RSA-FDH-VRF)	7
4.1.	RSA-FDH-VRF Proving	9
4.2.	RSA-FDH-VRF Proof to Hash	9
4.3.	RSA-FDH-VRF Verifying	10
4.4.	RSA-FDH-VRF Ciphersuites	11
5.	Elliptic Curve VRF (ECVRF)	11
5.1.	ECVRF Proving	14
5.2.	ECVRF Proof to Hash	15
5.3.	ECVRF Verifying	15
5.4.	ECVRF Auxiliary Functions	17
5.4.1.	ECVRF Encode to Curve	17
5.4.2.	ECVRF Nonce Generation	19
5.4.3.	ECVRF Challenge Generation	21
5.4.4.	ECVRF Decode Proof	21
5.4.5.	ECVRF Validate Key	22
5.5.	ECVRF Ciphersuites	24
6.	Implementation Status	26
7.	Security Considerations	27
7.1.	Key Generation	28
7.1.1.	Uniqueness and collision resistance with untrusted	

keys	28
7.1.2. Pseudorandomness with untrusted keys	28
7.2. Security Levels	28
7.3. Selective vs. Full Pseudorandomness	29
7.4. Proper pseudorandom nonce for ECVRF	30

7.5. Side-channel attacks	30
7.6. Proofs provide no secrecy for the VRF input	31
7.7. Prehashing	31
7.8. Hash function domain separation	31
7.9. Hash function salting	32
7.10. Futureproofing	32
8. Change Log	33
9. Contributors	34
10. References	34
10.1. Normative References	34
10.2. Informative References	36
Appendix A. Test Vectors for the ECVRFs	37
A.1. ECVRF-P256-SHA256-TAI	37
A.2. ECVRF-P256-SHA256-SSWU	38
A.3. ECVRF-EDWARDS25519-SHA512-TAI	40
A.4. ECVRF-EDWARDS25519-SHA512-ELL2	42
Authors' Addresses	44

[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 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 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 structure.

[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

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

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 private 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 = 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 π_1 and π_2 such that $\text{VRF_verify}(\text{PK}, \alpha, \pi_1)$ outputs (VALID, β_1), $\text{VRF_verify}(\text{PK}, \alpha, \pi_2)$ outputs (VALID, β_2), and β_1 is not equal to β_2 .

For many applications, a slightly weaker security property called "trusted uniqueness" suffices. Trusted uniqueness is the same as full uniqueness, but it is guaranteed to hold only if the VRF keys PK and SK were generated in a trustworthy manner.

As further discussed in [Section 7.1.1](#), some VRFs specified in this document satisfy only trusted uniqueness, while others satisfy full uniqueness. VRFs in this document that satisfy only trusted uniqueness but not full uniqueness MUST NOT be used if the key generation process cannot be trusted.

[3.2.](#) Full Collision Resistance or Trusted Collision Resistance

Like any cryptographic hash function, VRFs need to be collision resistant. Collision 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 α_1 and α_2 that have the same VRF hash β , even if that adversary knows the private VRF key SK.

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

As further discussed in [Section 7.1.1](#), some VRFs specified in this document satisfy only trusted collision resistance, while others satisfy full collision resistance. VRFs in this document that satisfy only trusted collision resistance but not full collision resistance MUST NOT be used if the key generation process cannot be trusted.

[3.3.](#) Full Pseudorandomness or Selective Pseudorandomness

Pseudorandomness ensures that when an adversarial Verifier sees a VRF hash output β without its corresponding VRF proof π , then β 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 β (without its corresponding VRF proof π) on any adversarially-chosen "target" VRF input α 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 α' and observe their corresponding VRF hash outputs β' and proofs π' .

With "full pseudorandomness", the adversary is allowed to choose the "target" VRF input α at any time, even after it observes VRF outputs β' and proofs π' on a variety of chosen inputs α' .

"Selective pseudorandomness" is a weaker security property which suffices in many applications. Here, the adversary must choose the target VRF input α independently of the public VRF key PK , and before it observes VRF outputs β' and proofs π' on inputs α' 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 β 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 β from a random value by comparing β to the result of $VRF_hash(SK, \alpha)$.

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

Also, the VRF output β 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.](#) Some VRFs: Unpredictability Under Malicious Key Generation

As explained in [Section 3.3](#), pseudorandomness is guaranteed only if the VRF keys were generated in a trustworthy fashion. 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 uniform, no matter how the VRF keys are generated.

A formal definition of this property appears in Section 3.2 of [\[DGKR18\]](#). The RSA-FDH-VRF presented in this document does not satisfy this property. The ECVRF presented in this document satisfies this property if `validate_key` parameter given to the `ECVRF_verify` is `TRUE`.

[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)

RSVP1 – 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

`RSAFDHVRF_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:

Internet-Draft

VRF

February 2022

RSAFDHVRF_proof_to_hash should be run only on pi_string that is known to have been produced by RSAFDHVRP_prove, or from within RSAFDHVRP_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

RSAFDHVRP_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 = OS2IP(pi_string)`
2. `m = RSAVP1((n, e), s)`; if `RSAVP1` returns "signature representative out of range", output "INVALID" and stop.
3. `mgf_domain_separator = 0x01`

4. `EM' = MGF1(suite_string || mgf_domain_separator || MGF_salt || alpha_string, k - 1)`
5. `m' = OS2IP(EM')`
6. If `m` and `m'` are equal, output ("VALID", `RSAFDHVRF_proof_to_hash(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 = I2OSP(k, 4) || 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 = I2OSP(k, 4) || 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 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

$Y = x \cdot B$ - 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 private 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. $\text{pi_string} = \text{point_to_string}(\text{Gamma}) || \text{int_to_string}(c, cLen) || \text{int_to_string}(s, qLen)$
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 is supported.

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"

Internet-Draft

VRF

February 2022

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 $hLen \times 8$

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]...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 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' = 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.

Also note that if the cofactor is small, the total number of Y values that could cause Step 2 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 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 = 2707385501144840649318225287225658788936804267575313519
463743609750303402022
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 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} .

Internet-Draft

VRF

February 2022

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.)

[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 private 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 an 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 private 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 an 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/ietf-draft03.

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

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

7.1.1. Uniqueness and collision resistance with untrusted keys

The RSA-FDH-VRF satisfies the "trusted uniqueness" (see [Section 3.1](#)) and "trusted collision resistance" (see [Section 3.2](#)) properties as long as the VRF keys are generated correctly. 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 the standard); thus, RSA-FDH-VRF

defined in this document does not have "full uniqueness" and "full collision resistance". Therefore, if adversarial key generation is a concern, the RSA-FDH-VRF has to be enhanced by additional cryptographic checks 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" (see [Section 3.1](#)) and "trusted collision resistance" (see [Section 3.2](#)) properties. It additionally satisfies "full collision resistance" if `validate_key` parameter given to the `ECVRF_verify` is TRUE.

[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](#). 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 trusted uniqueness: 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 3.3](#) 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. Such applications may consider the following two options:

- * 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.

- * 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 private VRF key x (and thus break pseudorandomness of the VRF) after observing several valid VRF proofs π_i . 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 private 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 α 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 α .

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 α is secret and is hashed by the VRF on-the-fly. This is because the algorithm's running time depends on the VRF input α , and thus creates a timing channel that can be used to learn information about α . 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 α , following recommendations in [\[I-D.irtf-cfrg-hash-to-curve\]](#).

Internet-Draft

VRF

February 2022

[7.6.](#) Proofs provide no secrecy for the VRF input

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

[7.7.](#) Prehashing

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

[7.8.](#) Hash function domain separation

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 `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 queries made to the hash function for a given SK and α are overwhelmingly unlikely to equal each other or to any queries made to the hash function for the same SK and different α . 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.

9. Contributors

This document also 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, David C. Lawrence, Derek Ting-Haye Leung, Antonio Marcedone, Piotr Nojszewski, Chris Peikert, Trevor Perrin, Sam Scott, Stanislav Smyshlyaev, Adam Suhl, Nick Sullivan, Christopher Wood, Jiayu Xu, and Annie Yousar provided valuable input to this draft. Riad Wahby helped this document align with [draft-irtf-cfrg-hash-to-curve](#).

10. References

10.1. Normative References

Goldberg, et al. Expires 10 August 2022 [Page 34]

Internet-Draft VRF February 2022

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", [BCP 14](#), [RFC 2119](#), DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", [RFC 8017](#), DOI 10.17487/RFC8017, November 2016, <<https://www.rfc-editor.org/info/rfc8017>>.
- [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-13](#), 10 November 2021, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hash-to-curve-13>>.
- [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.
- [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](https://datatracker.ietf.org/doc/html/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>>.

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

A.1. ECVRF-P256-SHA256-TAI

The example secret keys and messages in Examples 1 and 2 are taken from [Appendix A.2.5 of \[RFC6979\]](#).

Example 1:

```
SK = x =
c9afa9d845ba75166b5c215767b1d6934e50c3db36e89b127b8a622b120f6721
PK =
0360fed4ba255a9d31c961eb74c6356d68c049b8923b61fa6ce669622e60f29fb6
alpha = 73616d706c65 (ASCII "sample")
try_and_increment succeeded on ctr = 1
H =
0272a877532e9ac193aff4401234266f59900a4a9e3fc3cfc6a4b7e467a15d06d4
k =
0d90591273453d2dc67312d39914e3a93e194ab47a58cd598886897076986f77
U = k*B =
02bb6a034f67643c6183c10f8b41dc4babf88bffa154b674e377d90bde009c21672
V = k*H =
02893ebee7af9a0faa6da810da8a91f9d50e1dc071240c9706726820ff919e8394
pi = 035b5c726e8c0e2c488a107c600578ee75cb702343c153cb1eb8dec77f4b5
071b4a53f0a46f018bc2c56e58d383f2305e0975972c26feea0eb122fe7893c15a
f376b33edf7de17c6ea056d4d82de6bc02f
beta =
a3ad7b0ef73d8fc6655053ea22f9bede8c743f08bbbed3d38821f0e16474b505e
```

Example 2:

```
SK = x =
c9afa9d845ba75166b5c215767b1d6934e50c3db36e89b127b8a622b120f6721
PK =
0360fed4ba255a9d31c961eb74c6356d68c049b8923b61fa6ce669622e60f29fb6
alpha = 74657374 (ASCII "test")
try_and_increment succeeded on ctr = 3
H =
02173119b4fff5e6f8afed4868a29fe8920f1b54c2cf89cc7b301d0d473de6b974
k =
5852353a868bdce26938cde1826723e58bf8cb06dd2fed475213ea6f3b12e961
U = k*B =
022779a2cafc6b65414c4a04a4b4d2adf4c50395f57995e89e6de823250d91bc48e
V = k*H =
033b4a14731672e82339f03b45ff6b5b13dee7ada38c9bf1d6f8f61e2ce5921119
```

```
pi = 034dac60aba508ba0c01aa9be80377ebd7562c4a52d74722e0abae7dc3080
ddb56c19e067b15a8a8174905b13617804534214f935b94c2287f797e393eb0816
969d864f37625b443f30f1a5a33f2b3c854
beta =
a284f94ceec2ff4b3794629da7cbafa49121972671b466cab4ce170aa365f26d
```

The example secret key in Example 3 is taken from [Appendix L.4.2](#) of [\[ANSI.X9-62-2005\]](#).

Example 3:

```
SK = x =
2ca1411a41b17b24cc8c3b089cfd033f1920202a6c0de8abb97df1498d50d2c8
PK =
03596375e6ce57e0f20294fc46bdfcfd19a39f8161b58695b3ec5b3d16427c274d
alpha = 4578616d706c65207573696e67204543445341206b65792066726f6d20
417070656e646978204c2e342e32206f6620414e53492e58392d36322d32303035
(ASCII "Example using ECDSA key from Appendix L.4.2 of
ANSI.X9-62-2005")
try_and_increment succeeded on ctr = 1
H =
0258055c26c4b01d01c00fb57567955f7d39cd6f6e85fd37c58f696cc6b7aa761d
k =
5689e2e08e1110b4dda293ac21667eac6db5de4a46a519c73d533f69be2f4da3
U = k*B =
020f465cd0ec74d2e23af0abde4c07e866ae4e5138bde5dd1196b8843f380db84
V = k*H =
036cb6f811428fc4904370b86c488f60c280fa5b496d2f34ff8772f60ed24b2d1d
pi = 03d03398bf53aa23831d7d1b2937e005fb0062cbefa06796579f2a1fc7e7b
8c667d091c00b0f5c3619d10ecea44363b5a599cad5b2957e223fec62e81f7b48
25fc799a771a3d7334b9186bdbbee87316b1
beta =
90871e06da5caa39a3c61578ebb844de8635e27ac0b13e829997d0d95dd98c19
```

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

The example secret keys and messages in Examples 4 and 5 are taken from [Appendix A.2.5 of \[RFC6979\]](#).

Example 4:

```
SK = x =
c9afa9d845ba75166b5c215767b1d6934e50c3db36e89b127b8a622b120f6721
PK =
0360fed4ba255a9d31c961eb74c6356d68c049b8923b61fa6ce669622e60f29fb6
alpha = 73616d706c65 (ASCII "sample")
```

In SSWU: uniform_bytes = 5024e98d6067dec313af09ff0cbe78218324a645c
2a4b0aae2453f6fe91aa3bd9471f7b4a5fbf128e4b53f0c59603f7e

In SSWU: u =
df565615a2372e8b31b8771f7503bafc144e48b05688b97958cc27ce29a8d810
In SSWU: x1 =
e7e39eb8a4c982426fcff629e55a3e13516cf62c02c369b1e750316f5e94eb
In SSWU: gx1 is a nonsquare
H =
02b31973e872d4a097e2cfae9f37af9f9d73428fde74ac537dda93b5f18dbc5842
k =
e92820035a0a8afe132826c6312662b6ea733fc1a0d33737945016de54d02dd8
U = k*B =
031490f49d0355ffcdf66e40df788bee93861917ee713acff79be40d20cc91a30a
V = k*H =
03701df0228138fa3d16612c0d720389326b3265151bc7ac696ea4d0591cd053e3
pi = 0331d984ca8fece9cbb9a144c0d53df3c4c7a33080c1e02ddb1a96a365394
c7888782ffffde7b842c38c20c08de6ec6c2e7027a97000f2c9fa4425d5c03e639f
b48fde58114d755985498d7eb234cf4aed9
beta =
21e66dc9747430f17ed9efeda054cf4a264b097b9e8956a1787526ed00dc664b

Example 5:

SK = x =
c9afa9d845ba75166b5c215767b1d6934e50c3db36e89b127b8a622b120f6721
PK =
0360fed4ba255a9d31c961eb74c6356d68c049b8923b61fa6ce669622e60f29fb6
alpha = 74657374 (ASCII "test")
In SSWU: uniform_bytes = 910cc66d84a57985a1d15843dad83fd9138a109af
b243b7fa5d64d766ec9ca3894fdcf46eb21a3972eb452a4232fd3
In SSWU: u =
d8b0107f7e7aa36390240d834852f8703a6dc407019d6196bda5861b8fc00181
In SSWU: x1 =
ccc747fa7318b9486ce4044adbbecaa084c27be6eda88eb7b7f3d688fd0968c7
In SSWU: gx1 is a square
H =
03ccc747fa7318b9486ce4044adbbecaa084c27be6eda88eb7b7f3d688fd0968c7
k =
febc3451ea7639fde2cf41ffd03f463124ecb3b5a79913db1ed069147c8a7dea
U = k*B =
031200f9900e96f811d1247d353573f47e0d9da601fc992566234fc1a5b37749ae

V = k*H =
02d3715dcfee136c7ae50e95ffca76f4ca6c29ddfb92a39c31a0d48e75c6605cd1
pi = 03f814c0455d32dbc75ad3aea08c7e2db31748e12802db23640203aebf1fa
8db2743aad348a3006dc1caad7da28687320740bf7dd78fe13c298867321ce3b36
b79ec3093b7083ac5e4daf3465f9f43c627
beta =
8e7185d2b420e4f4681f44ce313a26d05613323837da09a69f00491a83ad25dd

The example secret key in Example 6 is taken from [Appendix L.4.2](#) of [\[ANSI.X9-62-2005\]](#).

Example 6:

SK = x =
2ca1411a41b17b24cc8c3b089cfd033f1920202a6c0de8abb97df1498d50d2c8
PK =
03596375e6ce57e0f20294fc46bdfcfd19a39f8161b58695b3ec5b3d16427c274d
alpha = 4578616d706c65207573696e67204543445341206b65792066726f6d20
417070656e646978204c2e342e32206f6620414e53492e58392d36322d32303035
(ASCII "Example using ECDSA key from [Appendix L.4.2](#) of
ANSI.X9-62-2005")
In SSWU: uniform_bytes = 9b81d55a242d3e8438d3bcfb1bee985a87fd14480
2c9268cf9adeee160e6e9ff765569797a0f701cb4316018de2e7dd4
In SSWU: u =
e43c98c2ae06d13839fedb0303e5ee815896beda39be83fb11325b97976efdce
In SSWU: x1 =
be9e195a50f175d3563aed8dc2d9f513a5536c1e9aee1757d86c08d32d582a86
In SSWU: gx1 is a nonsquare
H =
022dd5150e5a2a24c66feab2f68532be1486e28e07f1b9a055cf38ccc16f6595ff
k =
8e29221f33564f3f66f858ba2b0c14766e1057adbd422c3e7d0d99d5e142b613
U = k*B =
03a8823ff9fd16bf879261c740b9c7792b77fee0830f21314117e441784667958d
V = k*H =
02d48fbb45921c755b73b25be2f23379e3ce69294f6cee9279815f57f4b422659d
pi = 039f8d9cdc162c89be2871cbcb1435144739431db7fab437ab7bc4e2651a9
e99d5488405a11a6c7fc8defddd9e1573a563b7333aab4effe73ae9803274174c6
59269fd39b53e133dcd9e0d24f01288de9
beta =

4fbadf33b42a5f42f23a6f89952d2e634a6e3810f15878b46ef1bb85a04fe95a

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

The example secret keys and messages in Examples 7, 8, and 9 are taken from [Section 7.1 of \[RFC8032\]](#).

Example 7:

```
SK =
9d61b19def fd5a60ba844af492ec2cc44449c5697b326919703bac031cae7f60
PK =
d75a980182b10ab7d54bfed3c964073a0ee172f3daa62325af021a68f707511a
alpha = (the empty string)
x =
307c83864f2833cb427a2ef1c00a013cfdff2768d980c0a3a520f006904de94f
```

```
try_and_increment succeeded on ctr = 0
H =
91bbbed02a99461df1ad4c6564a5f5d829d0b90cfc7903e7a5797bd658abf3318
k = 7100f3d9eadb6dc4743b029736ff283f5be494128df128df2817106f345b85
94b6d6da2d6fb0b4c0257eb337675d96eab49cf39e66cc2c9547c2bf8b2a6afae4
U = k*B =
aef27c725be964c6a9bf4c45ca8e35df258c1878b838f37d9975523f09034071
V = k*H =
5016572f71466c646c119443455d6cb9b952f07d060ec8286d678615d55f954f
pi = 8657106690b5526245a92b003bb079ccd1a92130477671f6fc01ad16f26f7
23f26f8a57ccaed74ee1b190bed1f479d9727d2d0f9b005a6e456a35d4fb0daab1
268a1b0db10836d9826a528ca76567805
beta = 90cf1df3b703cce59e2a35b925d411164068269d7b2d29f3301c03dd757
876ff66b71dda49d2de59d03450451af026798e8f81cd2e333de5cdf4f3e140fdd
8ae
```

Example 8:

```
SK =
4ccd089b28ff96da9db6c346ec114e0f5b8a319f35aba624da8cf6ed4fb8a6fb
PK =
3d4017c3e843895a92b70aa74d1b7ebc9c982ccf2ec4968cc0cd55f12af4660c
alpha = 72 (1 byte)
x =
68bd9ed75882d52815a97585caf4790a7f6c6b3b7f821c5e259a24b02e502e51
```

```
try_and_increment succeeded on ctr = 1
H =
5b659fc3d4e9263fd9a4ed1d022d75eaacc20df5e09f9ea937502396598dc551
k = 42589bbf0c485c3c91c1621bb4bfe04aed7be76ee48f9b00793b2342acb9c1
67cab856f9f9d4febc311330c20b0a8afd3743d05433e8be8d32522ecdc16cc5ce
U = k*B =
1dcb0a4821a2c48bf53548228b7f170962988f6d12f5439f31987ef41f034ab3
V = k*H =
fd03c0bf498c752161bae4719105a074630a2aa5f200ff7b3995f7bfb1513423
pi = f3141cd382dc42909d19ec5110469e4feae18300e94f304590abdc48aed
5933bf0864a62558b3ed7f2fea45c92a465301b3bbf5e3e54ddf2d935be3b67926
da3ef39226bbc355bdc9850112c8f4b02
beta = eb4440665d3891d668e7e0fc587f1b4bd7fbfe99d0eb2211ccec90496
310eb5e33821bc613efb94db5e5b54c70a848a0bef4553a41befc57663b56373a5
031
```

Example 9:

```
SK =
c5aa8df43f9f837bedb7442f31dcb7b166d38535076f094b85ce3a2e0b4458f7
PK =
fc51cd8e6218a1a38da47ed00230f0580816ed13ba3303ac5deb911548908025
alpha = af82 (2 bytes)
```

```
x =
909a8b755ed902849023a55b15c23d11ba4d7f4ec5c2f51b1325a181991ea95c
try_and_increment succeeded on ctr = 0
H =
bf4339376f5542811de615e3313d2b36f6f53c0acfebb482159711201192576a
k = 38b868c335ccda94a088428cbf3ec8bc7955bfaffe1f3bd2aa2c59fc31a0fe
bc59d0e1af3715773ce11b3bbdd7aba8e3505d4b9de6f7e4a96e67e0d6bb6d6c3a
U = k*B =
2bae73e15a64042fceb062abe7e432b2eca6744f3e8265bc38e009cd577ecd5
V = k*H =
88cba1cb0d4f9b649d9a86026b69de076724a93a65c349c988954f0961c5d506
pi = 9bc0f79119cc5604bf02d23b4caede71393cedfbb191434dd016d30177ccb
f8096bb474e53895c362d8628ee9f9ea3c0e52c7a5c691b6c18c9979866568add7
a2d41b00b05081ed0f58ee5e31b3a970e
beta = 645427e5d00c62a23fb703732fa5d892940935942101e456ecca7bb217c
61c452118fec1219202a0edcf038bb6373241578be7217ba85a2687f7a0310b2df
19f
```

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

The example secret keys and messages in Examples 10, 11, and 12 are taken from [Section 7.1 of \[RFC8032\]](#).

Example 10:

```
SK =
9d61b19deffd5a60ba844af492ec2cc44449c5697b326919703bac031cae7f60
PK =
d75a980182b10ab7d54bfed3c964073a0ee172f3daa62325af021a68f707511a
alpha = (the empty string)
x =
307c83864f2833cb427a2ef1c00a013cfdff2768d980c0a3a520f006904de94f
In Elligator2: uniform_bytes = d620782a206d9de584b74e23ae5ee1db5ca
5298b3fc527c4867f049dee6dd419b3674967bd614890f621c128d72269ae
In Elligator2: u =
30f037b9745a57a9a2b8a68da81f397c39d46dee9d047f86c427c53f8b29a55c
In Elligator2: gx1 =
8cb66318fb2cea01672d6c27a5ab662ae33220961607f69276080a56477b4a08
In Elligator2: gx1 is a square
H =
b8066ebbb706c72b64390324e4a3276f129569eab100c26b9f05011200c1bad9
k = b5682049fee54fe2d519c9afff73bbfad724e69a82d5051496a42458f817be
d7a386f96b1a78e5736756192aeb1818a20efb336a205ffede351cfe88dab8d41c
U = k*B =
762f5c178b68f0cddcc1157918edf45ec334ac8e8286601a3256c3bbf858edd9
V = k*H =
4652eba1c4612e6fce762977a59420b451e12964adbe4fbecd58a7aef55860af
```

```
pi = 7d9c633ffeee27349264cf5c667579fc583b4bda63ab71d001f89c10003ab
46f14adf9a3cd8b8412d9038531e865c341cafa73589b023d14311c331a9ad15ff
2fb37831e00f0acaa6d73bc9997b06501
beta = 9d574bf9b8302ec0fc1e21c3ec5368269527b87b462ce36dab2d14ccf80
c53cccf6758f058c5b1c856b116388152bbe509ee3b9ecfe63d93c3b4346c1fbc6
c54
```

Example 11:

```
SK =
4ccd089b28ff96da9db6c346ec114e0f5b8a319f35aba624da8cf6ed4fb8a6fb
```



```

PK =
3d4017c3e843895a92b70aa74d1b7ebc9c982ccf2ec4968cc0cd55f12af4660c
alpha = 72 (1 byte)
x =
68bd9ed75882d52815a97585caf4790a7f6c6b3b7f821c5e259a24b02e502e51
In Elligator2: uniform_bytes = 04ae20a9ad2a2330fb33318e376a2448bd7
7bb99e81d126f47952b156590444a9225b84128b66a2f15b41294fa2f2f6d
In Elligator2: u =
3092f033b16d4d5f74a3f7dc7091fe434b449065152b95476f121de899bb773d
In Elligator2: gx1 =
25d7fe7f82456e7078e99fdb24ef2582b4608357cdba9c39a8d535a3fd98464d
In Elligator2: gx1 is a nonsquare
H =
76ac3ccb86158a9104dff819b1ca293426d305fd76b39b13c9356d9b58c08e57
k = 88bf479281fd29a6cbdf6d67e2c5ec0024d92f14eae58f43f22f37c4c37f1
d41e65c036fbf01f9fba11d554c07494d0c02e7e5c9d64be88ef78cab7544e444d
U = k*B =
8ec26e77b8cb3114dd2265fe1564a4efb40d109aa3312536d93dfe3d8d80a061
V = k*H =
fe799eb5770b4e3a5a27d22518bb631db183c8316bb552155f442c62a47d1c8b
pi = 47b327393ff2dd81336f8a2ef10339112401253b3c714eeda879f12c50907
2ef055b48372bb82efbdce8e10c8cb9a2f9d60e93908f93df1623ad78a86a028d6
bc064dbfc75a6a57379ef855dc6733801
beta = 38561d6b77b71d30eb97a062168ae12b667ce5c28caccd76bc88e093e4
635987cd96814ce55b4689b3dd2947f80e59aac7b7675f8083865b46c89b2ce9cc
735

```

Example 12:

```

SK =
c5aa8df43f9f837bedb7442f31dcb7b166d38535076f094b85ce3a2e0b4458f7
PK =
fc51cd8e6218a1a38da47ed00230f0580816ed13ba3303ac5deb911548908025
alpha = af82 (2 bytes)
x =
909a8b755ed902849023a55b15c23d11ba4d7f4ec5c2f51b1325a181991ea95c

```

```

In Elligator2: uniform_bytes = be0aed556e36cdfddf8f1eeddbb7356a24f
ad64cf95a922a098038f215588b216beabbfe6acf20256188e883292b7a3a
In Elligator2: u =
f6675dc6d17fc790d4b3f1c6acf689a13d8b5815f23880092a925af94cd6fa24

```

In Elligator2: $gx1 =$
a63d48e3247c903e22fd9b88fd9295e396712a5fe576af335dbe16f99f0af26c
In Elligator2: $gx1$ is a square
 $H =$
13d2a8b5ca32db7e98094a61f656a08c6c964344e058879a386a947a4e189ed1
 $k =$ a7ddd74a3a7d165d511b02fa268710ddbb3b939282d276fa2efcfa5aaf79cf
576087299ca9234aacd7cd674d912deba00f4e291733ef189a51e36c861b3d683b
 $U = k*B =$
a012f35433df219a88ab0f9481f4e0065d00422c3285f3d34a8b0202f20bac60
 $V = k*H =$
fb613986d171b3e98319c7ca4dc44c5dd8314a6e5616c1a4f16ce72bd7a0c25a
 $\pi =$ 926e895d308f5e328e7aa159c06eddb56d06846abf5d98c2512235eaa57f
dce35b46edfc655bc828d44ad09d1150f31374e7ef73027e14760d42e77341fe05
467bb286cc2c9d7fde29120a0b2320d04
 $\beta =$ 121b7f9b9aaaa29099fc04a94ba52784d44eac976dd1a3cca458733be5c
d090a7b5fbd148444f17f8daf1fb55cb04b1ae85a626e30a54b4b0f8abf4a43314
a58

Authors' Addresses

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

Email: goldbe@cs.bu.edu

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

Email: reyzin@bu.edu

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

Email: dipapado@cse.ust.hk

Internet-Draft

VRF

February 2022

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

Email: jvcelak@ns1.com

