

Verifiable Random Functions (VRFs)

Sharon Goldberg, Jan Vcelak, Dimitrios Papadopoulos, Leonid Reyzin. "Verifiable Random Functions (VRFs)."

<https://hdl.handle.net/2144/29225>

"Downloaded from OpenBU. Boston University's institutional repository."

CFRG
Internet-Draft
Intended status: Standards Track
Expires: September 6, 2018

S. Goldberg
L. Reyzin
Boston University
D. Papadopoulos
Hong Kong University of Science and Technology
J. Vcelak
NSI
March 5, 2018

Verifiable Random Functions (VRFs)
draft-irtf-cfrg-vrf-01

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 Elliptic Curves (EC).

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 September 6, 2018.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to [BCP 78](#) and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of

publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1.	Introduction	3
1.1.	Rationale	3
1.2.	Requirements	3
1.3.	Terminology	3
2.	VRF Algorithms	4
3.	VRF Security Properties	4
3.1.	Full Uniqueness or Trusted Uniqueness	4
3.2.	Full Collision Resistance or Trusted Collision Resistance	5
3.3.	Full Pseudorandomness or Selective Pseudorandomness	5
3.4.	An additional pseudorandomness property	6
4.	RSA Full Domain Hash VRF (RSA-FDH-VRF)	7
4.1.	RSA-FDH-VRF Proving	8
4.2.	RSA-FDH-VRF Proof To Hash	8
4.3.	RSA-FDH-VRF Verifying	9
5.	Elliptic Curve VRF (EC-VRF)	9
5.1.	EC-VRF Proving	11
5.2.	EC-VRF Proof To Hash	11
5.3.	EC-VRF Verifying	12
5.4.	EC-VRF Auxiliary Functions	13
5.4.1.	EC-VRF Hash To Curve	13
5.4.2.	EC-VRF Hash Points	14
5.4.3.	EC-VRF Decode Proof	15
5.5.	EC-VRF Ciphersuites	15
5.6.	When the EC-VRF Keys are Untrusted	17
5.6.1.	EC-VRF Validate Key	17
6.	Implementation Status	18
7.	Security Considerations	18
7.1.	Key Generation	18
7.1.1.	Uniqueness and collision resistance with untrusted keys	18
7.1.2.	Pseudorandomness with untrusted keys	19
7.2.	Selective vs Full Pseudorandomness	19
7.3.	Proper randomness for EC-VRF	20
7.4.	Timing attacks	20
8.	Change Log	20
9.	Contributors	20
10.	References	21
10.1.	Normative References	21
10.2.	Informative References	21

Authors' Addresses	22
------------------------------	----

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 secret key and uses the VRF hashing to construct a hash-based data structure on the input data. Due to the nature of the VRF, only the Prover can answer queries about whether or not some data is stored in the data structure. Anyone who knows the 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: The input to be hashed by the VRF.

beta: The VRF hash output.

pi: 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.

A VRF 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 a pair of inputs (SK, alpha). The private key SK is also used 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 as

$$\text{beta} = \text{VRF_proof2hash}(\text{pi})$$

Notice that this means that

$$\text{VRF_hash}(\text{SK}, \text{alpha}) = \text{VRF_proof2hash}(\text{VRF_prove}(\text{SK}, \text{alpha}))$$

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

$$\text{VRF_verify}(\text{PK}, \text{alpha}, \text{pi})$$

that outputs VALID if $\text{beta} = \text{VRF_proof2hash}(\text{pi})$ is correct VRF hash of alpha under key PK, and outputs 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 secret key SK.

"Full uniqueness" states that a computationally-bounded adversary cannot choose a VRF public key PK, a VRF input alpha, two different

VRF hash outputs β_1 and β_2 , and two proofs π_1 and π_2 such that $\text{VRF_verify}(PK, \alpha, \pi_1)$ and $\text{VRF_verify}(PK, \alpha, \pi_2)$ both output VALID.

A slightly weaker security property called "trusted uniqueness" suffices 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 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 secret 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 secret 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 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 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" and `beta = VRF_proof2hash(pi)`.

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. An additional pseudorandomness property

[TODO: The following property is not needed for applications that use VRFs to prevent enumeration of hash-based data structures. However, we noticed that some other applications of VRF rely on this property. As we have not yet found a formal definition of this property in the literature, we write it down here.]

Pseudorandomness, as defined in [Section 3.3](#), does not hold if the VRF keys were generated adversarially.

There is, however, a different type of pseudorandomness that could hold even if the VRF keys are generated adversarially, as long as the VRF input alpha is unpredictable. Suppose the VRF keys are generated by an adversary. Then, a VRF hash output beta should look pseudorandom to the adversary as long as (1) its corresponding VRF hash alpha is chosen randomly and independently of the VRF key, (2) alpha is unknown to the adversary, (3) the corresponding proof pi is unknown to the adversary, and (4) the VRF public key chosen by the adversary is valid.

[TODO: It should be possible to get the EC-VRF to satisfy this property, as long as verifiers run an `VRF_validate_key()` key function upon receipt of VRF public keys. However, we need to work out exactly what properties are needed from the VRF public keys in order for this property to hold. Some additional checks might need to be added to the `ECVRF_validate_key()` function. Need to work out what are these checks.]

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 [[nsec5ecc](#)].

The VRF computes the proof π as a deterministic RSA signature on input α 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 β is simply obtained by hashing the proof π 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

Fixed options:

Hash - cryptographic hash function

$hLen$ - output length in octets of hash function Hash

Constraints on options:

Cryptographic security of Hash is at least as high as the cryptographic security level of the RSA key

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 \[RFC8017\]](#)

RSAPV1 - RSA verification primitive as defined in [Section 5.2.2 of \[RFC8017\]](#)

MGF1 - Mask Generation Function based on a hash function as defined in Section B.2.1 of [\[RFC8017\]](#)

4.1. RSA-FDH-VRF Proving

RSAFDHVRF_prove(K, alpha)

Input:

K - RSA private key

alpha - VRF hash input, an octet string

Output:

pi - proof, an octet string of length k

Steps:

1. EM = MGF1(alpha, k - 1)
2. m = OS2IP(EM)
3. s = RSASP1(K, m)
4. pi = I2OSP(s, k)
5. Output pi

4.2. RSA-FDH-VRF Proof To Hash

RSAFDHVRF_proof2hash(pi)

Input:

pi - proof, an octet string of length k

Output:

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

Steps:

1. $\beta = \text{Hash}(\pi)$
2. Output β

4.3. RSA-FDH-VRF Verifying

`RSAFDHVRF_verify((n, e), alpha, pi)`

Input:

(n, e) - RSA public key

α - VRF hash input, an octet string

π - proof to be verified, an octet string of length n

Output:

"VALID" or "INVALID"

Steps:

1. $s = \text{OS2IP}(\pi)$
2. $m = \text{RSAP1}((n, e), s)$
3. $EM = \text{I2OSP}(m, k - 1)$
4. $EM' = \text{MGF1}(\alpha, k - 1)$
5. If EM and EM' are equal, output "VALID"; else output "INVALID".

5. Elliptic Curve VRF (EC-VRF)

The Elliptic Curve Verifiable Random Function (EC-VRF) 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 [[nsec5ecc](#)].

Fixed options:

F - finite field

$2n$ - length, in octets, of a field element in F

E - elliptic curve (EC) defined over F

m - 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

cofactor - number of points on E divided by q

g - generator of group G

Hash - cryptographic hash function

$hLen$ - output length in octets of Hash

Constraints on options:

Field elements in F have bit lengths divisible by 16

$hLen$ is equal to $2n$

Parameters used:

$y = g^x$ - VRF public key, an EC point

x - VRF private key, an integer where $0 < x < q$

Notation and primitives used:

p^k - when p is an EC point: point multiplication, i.e. k repetitions of group operation on EC point p . when p is an integer: exponentiation

$||$ - octet string concatenation

I2OSP - nonnegative integer conversion to 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 \[RFC8017\]](#)

EC2OSP - conversion of EC point to an m -octet string as specified in [Section 5.5](#)

OS2ECP - conversion of an m -octet string to EC point as specified in [Section 5.5](#). OS2ECP returns INVALID if the octet string does not convert to a valid EC point.

RS2ECP - conversion of a random $2n$ -octet string to an EC point as specified in [Section 5.5](#)

5.1. EC-VRF Proving

Note: this function is made more efficient by taking in the public VRF key y , as well as the private VRF key x .

ECVRF_prove(y , x , α)

Input:

y - public key, an EC point

x - private key, an integer

α - VRF input, an octet string

Output:

π - VRF proof, octet string of length $m+3n$

Steps:

1. $h = \text{ECVRF_hash_to_curve}(y, \alpha)$
2. $\gamma = h^x$
3. choose a random integer nonce k from $[0, q-1]$
4. $c = \text{ECVRF_hash_points}(g, h, y, \gamma, g^k, h^k)$
5. $s = k - c \cdot x \pmod q$ (where $*$ denotes integer multiplication)
6. $\pi = \text{EC2OSP}(\gamma) \parallel \text{I2OSP}(c, n) \parallel \text{I2OSP}(s, 2n)$
7. Output π

5.2. EC-VRF Proof To Hash

ECVRF_proof2hash(π)

Input:

π - VRF proof, octet string of length $m+3n$

Output:

"INVALID", or

beta - VRF hash output, octet string of length $2n$

Steps:

1. $D = \text{ECVRF_decode_proof}(\text{pi})$
2. If D is "INVALID", output "INVALID" and stop
3. $(\text{gamma}, c, s) = D$
4. $\text{beta} = \text{Hash}(\text{EC2OSP}(\text{gamma}^{\text{cofactor}}))$
5. Output beta

5.3. EC-VRF Verifying

$\text{ECVRF_verify}(y, \text{pi}, \text{alpha})$

Input:

y - public key, an EC point

pi - VRF proof, octet string of length $5n+1$

alpha - VRF input, octet string

Output:

"VALID" or "INVALID"

Steps:

1. $D = \text{ECVRF_decode_proof}(\text{pi})$
2. If D is "INVALID", output "INVALID" and stop
3. $(\text{gamma}, c, s) = D$
4. $u = y^c * g^s$ (where $*$ denotes EC point addition, i.e. a group operation on two EC points)
5. $h = \text{ECVRF_hash_to_curve}(y, \text{alpha})$
6. $v = \text{gamma}^c * h^s$ (where $*$ denotes EC point addition)
7. $c' = \text{ECVRF_hash_points}(g, h, y, \text{gamma}, u, v)$

8. If c and c' are equal, output "VALID"; else output "INVALID"

[TODO: We could hash alpha instead of gamma. Doing so costs more because alpha is longer, but may help salvage some security if ECVRF_hash_to_curve is broken and not collision-resistant. Need to analyze exact security preserved and decide if the tradeoff is worth it.]

5.4. EC-VRF Auxiliary Functions

[TODO: analyze whether domain separation for hash functions used here matters and if so how to ensure we have it.]

5.4.1. EC-VRF Hash To Curve

The ECVRF_hash_to_curve algorithm takes in an octet string alpha and converts it to h, an EC point in G.

5.4.1.1. ECVRF_hash_to_curve1

The following ECVRF_hash_to_curve1(y , alpha) 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. 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. See also [[Icart09](#)].

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

ECVRF_hash_to_curve1(y , alpha)

Input:

alpha - value to be hashed, an octet string

y - public key, an EC point

Output:

h - hashed value, a finite EC point in G

Steps:

1. $ctr = 0$

2. $pk = EC2OSP(y)$
3. $h = "INVALID"$
4. While h is "INVALID" or h is EC point at infinity:
 - A. $CTR = I2OSP(ctr, 4)$
 - B. $ctr = ctr + 1$
 - C. $attempted_hash = Hash(pk || alpha || CTR)$
 - D. $h = RS2ECP(attempted_hash)$
 - E. If h is not "INVALID" and $cofactor > 1$, set $h = h^{cofactor}$
5. Output h

5.4.1.2. ECVRF_hash_to_curve2

For applications where VRF input $alpha$ must be kept secret, the following `ECVRF_hash_to_curve` algorithm MAY be used as generic way to hash an octet string onto any elliptic curve.

[TODO: We should look into specifying the generic deterministic time `hash_to_curve` algorithm from [Icart09]. We should also consider Shallue-Woestijne-Ulas algorithm from [BCIMRT10] as well as Elligator (for Ed25519) and Elligator Squared for more general curves. Some of these options are summarized in an upcoming draft [SW18].]

5.4.2. EC-VRF Hash Points

`ECVRF_hash_points(p_1, p_2, ..., p_j)`

Input:

p_i - EC point in G

Output:

h - hash value, integer between 0 and $2^{(8n)-1}$

Steps:

1. $P =$ empty octet string
2. for p_i in $[p_1, p_2, \dots, p_j]$:
 $P = P || EC2OSP(p_i)$

3. $h1 = \text{Hash}(P)$
4. $h2 = \text{first } n \text{ octets of } h1$
5. $h = \text{OS2IP}(h2)$
6. Output h

5.4.3. EC-VRF Decode Proof

`ECVRF_decode_proof(pi)`

Input:

pi - VRF proof, octet string ($m+3n$ octets)

Output:

"INVALID", or

γ - EC point

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

s - integer between 0 and $2^{(16n)}-1$

Steps:

1. let γ' , c' , s' be pi split after m -th and $m+n$ -th octet
2. $\gamma = \text{OS2ECP}(\gamma')$
3. if $\gamma = \text{"INVALID"}$ output "INVALID" and stop.
4. $c = \text{OS2IP}(c')$
5. $s = \text{OS2IP}(s')$
6. Output γ , c , and s

5.5. EC-VRF Ciphersuites

This document defines EC-VRF-P256-SHA256 as follows:

- o The EC group G is the NIST-P256 elliptic curve, with curve parameters as specified in [[FIPS-186-3](#)] (Section D.1.2.3) and [[RFC5114](#)] ([Section 2.6](#)). For this group, $2n = 32$ and cofactor = 1.

- o The key pair generation primitive is specified in Section 3.2.1 of [SECG1].
- o EC2OSP is specified in Section 2.3.3 of [SECG1] with point compression on. This implies $m = 2n + 1 = 33$.
- o OS2ECP is specified in Section 2.3.4 of [SECG1].
- o $RS2ECP(h) = OS2ECP(0x02 || h)$. The input h 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].
- o The ECVRF_hash_to_curve function is as specified in Section 5.4.1.1.

This document defines EC-VRF-ED25519-SHA256 as follows:

- o The EC group G is the Ed25519 elliptic curve with parameters defined in Table 1 of [RFC8032]. For this group, $2n = 32$ and cofactor = 8.
- o The key pair generation primitive is specified in Section 5.1.5 of [RFC8032]
- o EC2OSP is specified in Section 5.1.2 of [RFC8032]. This implies $m = 2n = 32$.
- o OS2ECP is specified in Section 5.1.3 of [RFC8032].
- o RS2ECP is equivalent to OS2ECP.
- o The hash function Hash is SHA-256 as specified in [RFC6234].
- o The ECVRF_hash_to_curve function is as specified in Section 5.4.1.1.

[TODO: Should we add an EC-VRF-ED25519-SHA256-Eligator ciphersuite where the Eligator hash function is used for ECVRF_hash-to-curve? Should we add Eligator-Squared? We may want to consider options in the upcoming draft [SW18].]

[TODO: Add an Ed448 ciphersuite? This is probably not needed...]

[NOTE: In the unlikely case that future versions of this spec use a elliptic curve group G that does not also come with a specification of the group generator g , then we can still have full uniqueness and full collision resistance by adding an check to

ECVRF_validate_key(PK) that ensures that g is a point on the elliptic curve and g^{cofactor} is not the EC point at infinity.]

5.6. When the EC-VRF Keys are Untrusted

The EC-VRF as specified above is a VRF that satisfies the "trusted uniqueness", "trusted collision resistance", and "full pseudorandomness" properties defined in [Section 3](#). If the elliptic curve parameters (including the generator g) are trusted, but the VRF public key PK is not trusted, this VRF can be modified to additionally satisfy "full uniqueness", and "full collision resistance". This is done by additionally requiring the Verifier to perform the following validation procedure upon receipt of the public VRF key.

The Verifier MUST perform this validation procedure when the entity that generated the public VRF key is untrusted. The public key MUST NOT be used if this procedure returns "INVALID". Note well that this procedure is not sufficient if the elliptic curve E or if g , the generator of group G , is untrusted.

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

5.6.1. EC-VRF Validate Key

ECVRF_validate_key(PK)

Input:

PK - public key, an octet string

Output:

"INVALID", or

y - public key, an EC point

Steps:

1. $y = \text{OS2ECP}(PK)$
2. If y is "INVALID", output "INVALID" and stop
3. If y^{cofactor} is the EC point at infinity, output "INVALID" and stop

4. Output y

6. Implementation Status

An implementation of the RSA-FDH-VRF (SHA-256) and EC-VRF-P256-SHA256 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 EC-VRF implementation may be out of date as this spec has evolved.

The Key Transparency project at Google uses a VRF implementation that is similar to the EC-VRF-P256-SHA256, 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 EC-VRF is available at <<https://github.com/r2ishiguro/vrf>>.

An implementation similar to EC-VRF 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 EC-VRF-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 EC-VRF as specified in [Section 5.1-Section 5.5](#) satisfies 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 EC-VRF 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

EC-VRF with public key y satisfies the "full uniqueness" and "full collision resistance" properties.

The RSA-FDH-VRF satisfies 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 modified 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

[nsec5ecc] presents cryptographic reductions to an underlying hard problem (e.g. Decisional Diffie Hellman for the EC-VRF, or the standard RSA assumption for RSA-FDH-VRF) that prove the VRFs specified 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 quantitatively stronger security guarantees 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 EC-VRF, this means increasing the cryptographic strength of the EC group G . For both RSA-FDH-VRF

and EC-VRF the cryptographic strength of the hash function Hash may also potentially need to be increased.

7.3. Proper randomness for EC-VRF

Applications that use the EC-VRF defined in this document MUST ensure that the random nonce k used in the `ECVRF_prove` algorithm is chosen with proper randomness. 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 .

[TODO: In the next version of the draft we should add a specification for nonce generation.]

7.4. Timing attacks

The `EC-VRF_hash_to_curve` 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 `EC-VRF_hash_to_curve` 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.

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.

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 (Salesforce) and David C. Lawrence (Akamai) provided valuable input to this draft.

10. References

10.1. Normative References

- [FIPS-186-3]
National Institute for Standards and Technology, "Digital Signature Standard (DSS)", FIPS PUB 186-3, June 2009.
- [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>>.
- [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

- [BCIMRT10]
Brier, E., Coron, J., Icart, T., Madore, D., Randriam, H., and M. Tibouchi, "Efficient Indifferentiable Hashing into Ordinary Elliptic Curves", in CRYPTO, 2010.

- [I-D.vcelak-nsec5]
Vcelak, J., Goldberg, S., Papadopoulos, D., Huque, S., and D. Lawrence, "NSEC5, DNSSEC Authenticated Denial of Existence", [draft-vcelak-nsec5-06](#) (work in progress), January 2018.
- [Icart09] Icart, T., "How to Hash into Elliptic Curves", in CRYPTO, 2009.
- [MRV99] Michali, S., Rabin, M., and S. Vadhan, "Verifiable Random Functions", in FOCS, 1999.
- [nsec5ecc]
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>.
- [SW18] Sullivan, E. and C. Wood, "Hashing to Elliptic Curves", 2018.

Authors' Addresses

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

E-Mail: goldbe@cs.bu.edu

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

E-Mail: reyzin@bu.edu

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

E-Mail: dipapado@cse.ust.hkbu.edu

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

EMail: jvcelak@ns1.com