Bandersnatch VRFs

VRF

Definition: A *verifiable random function with auxiliary data (VRF-AD)* can be described with three functions:

- $VRF.KeyGen : () \mapsto (pk, sk)$ where pk is a public key and sk is its corresponding secret key.
- $VRF.Sign : (sk, msg, aux) \mapsto \sigma$ takes a secret key sk, an input msg, and auxiliary data aux, and then returns a VRF signature σ.
- $VRF.Eval : (sk, msg) \mapsto Out$ takes a secret key sk and an input msg, and then returns a VRF output Out.
- $VRF.Verify : (pk, msg, aux, \sigma) \mapsto (Out|prep)$ for a public key pk, an input msg, and auxiliary data aux, and then returns either an output Out or else failure $perp$.

Definition: For an elliptic curve E defined over finite field F with large prime subgroup G generated by point g, we call a VRF, EC-VRF is VRF-AD where $pk = sk.g$ and $VRF.Sign$ is an elliptic curve signature scheme.

All VRFs described in this specification are EC-VRF. For input msg and aux auxilary data first we compute the $VRFInput$ which is a point on elliptic curve E as follows:

$$t \leftarrow Transcript(msg)$$

$$VRFiput := H2C(challange(t, "vrf - input")$$

where - $transcript$ function is described in [[ark-transcript]] section.
- $H2C : B \rightarrow G$ is a hash to curve function correspond to curve E specified in Section [[hash-to-curve]] for the specific choice of E

VRF Input

The VRF input ultimately is a point on the elliptic curve as out put of hash of the transcript using arkworks chosen hash for the given curve.

VRF Input point should always be created locally, either as a hash-to-cuve output of the transcripto or ocasionally some base point. It should never be sent over the wire nor deserialized???Do you mean serialized?

VRF Preoutput and Output

Definition: *VRF pre-output* is defined to be a point in E in serialized affine representation.

Definition: *VRF InOut* is defined as a pair as follows:

$$(VRFInput, VRFPreoutput)$$

Definition: *VRF output* is generated using VRF preoutput:

$$t \leftarrow Transcript(Domain)$$

$$append(t, "VrfOutput")$$

$$append(t, cofactor * VRFpreout)$$

$$VRFoutput \leftarrow t.challenge("")$$

VRF Key

Public key

A Public key of a VRF is a point on an Elliptic Curve E.
Public key is represented in Affine form and is serialized using Arkwork compressed serialized format.

IETF VRF

Refer to RFC-9381 for the details.

Bandersnatch Cipher Suite Configuration

Configuration follows the RFC-9381 suite specification guidelines.

- The EC group G is the Bandersnatch elliptic curve, in Twisted Edwards form, with the finite field and curve parameters as specified in the neuromancer standard curves database. For this group, `fLen` = `qLen` = 32 and `cofactor` = 4.
- The prime subgroup generator `g` is constructed following Zcash's guidelines: *"The generators of G1 and G2 are computed by finding the lexicographically smallest valid x-coordinate, and its lexicographically smallest y-coordinate and scaling it by the cofactor such that the result is not the point at infinity."*
  - g.x = `0x29c132cc2c0b34c5743711777bbe42f32b79c022ad998465e1e71866a252ae18`
  - g.y = `0x2a6c669eda123e0f157d8b50badcd586358cad81eee464605e3167b6cc974166`
- The public key generation primitive is `PK = SK · g`, with `SK` the secret key scalar and `g` the group generator. In this ciphersuite, the secret scalar `x` is equal to the secret key `SK`.
- `suite_string` = 0x33.
- `cLen` = 32.
- `encode_to_curve_salt` = `PK_string`.

- The `ECVRF_nonce_generation` function is as specified in Section 5.4.2.1 of RFC-9381.
- The `int_to_string` function encodes into the 32 bytes little endian representation.
- The `string_to_int` function decodes from the 32 bytes little endian representation.
- The point_to_string function converts a point on E to an octet string using compressed form. The Y coordinate is encoded using `int_to_string` function and the most significant bit of the last octet is used to keep track of the X's sign. This implies that the point is encoded on 32 bytes.
- The string_to_point function tries to decompress the point encoded according to `point_to_string` procedure. This function MUST outputs "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 5.4.1.2, with `h2c_suite_ID_string` = `"BANDERSNATCH_XMD:BLAKE2b_ELL2_RO_"`. The suite is defined in Section 8.5 of RFC9380.

Pedersen VRF

Pedersen VRF resembles EC VRF but replaces the public key by a Pedersen commitment to the secret key, which makes the Pedersen VRF useful in anonymized ring VRFs, or perhaps group VRFs.

Pedersen VRF

Strictly speaking Pederson VRF is not a VRF. Instead, it proves that the output has been generated with a secret key associated with a blinded public (instead of public key). The blinded public key is a cryptographic commitement to the public key. And it could unblinded to prove that the output of the VRF is corresponds to the public key of the signer.

Setup

PedersenVRF is initiated for prime subgroup $G < E$ of an elliptic curve E with $K, B \in G$ are defined to be *key base* and *blinding base* respectively.

PedersenVRF.Sign

Inputs:
- Transcript t of `ArkTranscript` type
- *input*: $VRFInput \in G$. - *sb*: Blinding coefficient $\in F$

- sk: A VRF secret key.
- pk: VRF verification key corresponds to sk.

Output:
- A Quintuple $(compk, KBrand, PORand, ks, bs)$2 corresponding to Pedersen-VRF signature

---

1. $AddLabel(t, "PedersenVRF")$
2. $compk = sk * G + sb * B$
3. AppendToTranscript("KeyCommitment")
4. AppendToTranscript(t, compk)
5. $krand \leftarrow RandomElement(F)$
6. $brand \leftarrow RandomElement(F)$
7. $KBrand \leftarrow krand * G + brand * B$
8. $PORand \leftarrow krand * input$
9. $AppendToTranscript(t, "PedersenR")$
10. $AppendToTranscript(t, "PedersenVrfChallenge")$
11. $c \rightarrow GetChallengeFromTranscript(t)$
12. $ks \rightarrow krand + sk * c$
13. $bs \rightarrow brand + c * sb$
14. **return** $(compk, KBrand, PORand, ks, bs)$

PedersenVRF.Verify

Inputs:
- t: Transcript of `ArkTranscript` type
- $input$: $VRFInput \in G$.
- $preout$: $VRFPreOutput \in G$.
- $(compk, KBrand, PORand, ks, bs)$ the quintuple results of PeredersonVRF.Sign

Output:
- True if Pedersen VRF signature verifys False otherwise.

Append$(t, "PedersenVRF")$
Append$(t, ""KeyCommitment")$
Append$(t, compk)$
$z1 \leftarrow PORand + c \times PreOut - In \times ks$ Append$(t, "PedersenR")$
Append$(t, KBrand||PORand)$
$c \leftarrow Challenge(t, "PedersenVrfChallenge")$

$z1 \leftarrow POrand + c \times preoutput - input \times ks$
$z1 \leftarrow ClearCofactor(z1)$
if $z1 \neq O$ **then return** False
$z2 \leftarrow KBrand + c \times compk - krand \times K - brand \times B$
$z2 \leftarrow ClearCofactor(z1)$
if $z2 \neq O$ **then return** False **else return** True

VRF input

Procedure to map arbitrary user input to a point follows the `hash_to_curve` procedure described by RFC9380.

`Suite_ID: "bandersnatch_XMD:SHA-512_ELL2_RO_"`

See ArkTranscript for details.

From transcript to point

You need to call challenge and add b"vrf-input" to it. getting random byte (some hash?) then hash to curve it.

Transcript

A Shake-128 based transcript construction which implements the Fiat-Shamir transform procedure.

We do basic domain separation using postfix writes of the lengths of written data (as opposed to the prefix writes by Merlin `TupleHash` from SP 800-185).

`H(item_1, item_2, ..., item_n)`

Represents the application of shake-128 to the concatenation of the serialization of each item followed by the serialization of the length of each objects, as a 32-bit unsigned integer.

```
bytes = encode(item_1) || encode(length(item_1)) || .. || encode(item_n) || encode(length(it
Shake128(bytes)
```

The length of each item should be less than 2^31.