

**saag@IETF'98**  
**Chicago, March 2017**

**draft-goldbe-vrf-00**

# **Verifiable Random Functions (VRF)**

**Sharon Goldberg (Boston University)**

**Dimitrios Papadopoulos (University of Maryland)**

**Jan Vcelak (ns1)**

**Contributors:** Leonid Reyzin (Boston University), Shumon Huque (Salesforce),  
David C. Lawrence (Akamai), Moni Naor & Asaf Ziv (Weizmann Institute)

# hash function zoo

---

## hash function:

SHA256

- no key
- hash =  $H(\text{input})$
- Verify: Check hash =  $H(\text{input})$

BLAKE

## pseudorandom function:

HMAC

- symmetric key  $k$
- hash =  $H(k, \text{input})$
- Verify: Cannot without  $k$

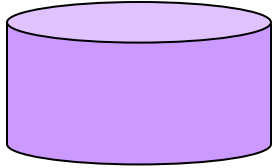
## verifiable random function (VRF):

- asymmetric key ( $SK, PK$ )
- hash =  $VRF\_hash(SK, \text{input})$
- Verify: Use  $PK$

# VRF: verifiable random function

---

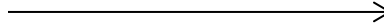
Verifier **PK**



Hasher **SK**



input



proof = **prove**(SK, input)

proof



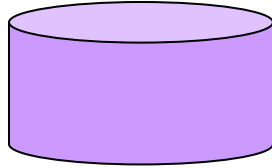
If **verify** (PK, input, proof)

hash = **proof2hash**(proof)

Else INVALID

# VRF security: trusted uniqueness

Verifier **PK**



Hasher **SK**



input



proof = **prove**(**SK**, input)

proof



If **verify** (**PK**, input, proof)

hash = **proof2hash**(proof)

Else INVALID

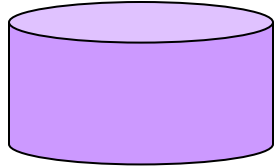
Like a regular hash function (eg SHA256)

Like a deterministic digital signature

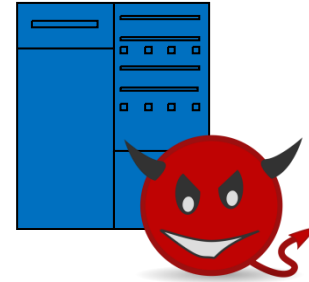
**1-to-1 relationship between input and hash. (As with SHA-256!)**

# VRF security: trusted uniqueness

Verifier **PK**



Hasher **SK**



input



proof



proof = **prove**(SK, input)

Like a deterministic digital signature

If **verify** (PK, input, proof)

hash = **proof2hash**(proof)

Else INVALID

Like a regular hash function (eg SHA256)

**1-to-1 relationship between input and hash. (As with SHA-256!)**

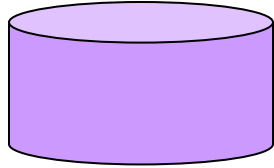
## Trusted uniqueness:

Suppose the VRF keys (**PK,SK**) are generated in a trusted way.

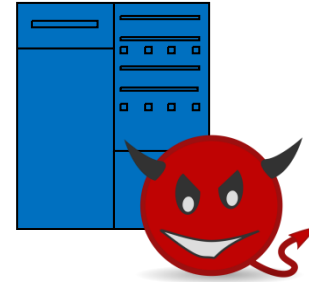
- If **PK** is fixed, then even an adversary that knows **SK** can't find
- ...two distinct VRF hash values that are valid for same input

# VRF security: trusted collision resistance

Verifier **PK**



Hasher **SK**



input



proof = **prove**(SK, input)

proof



If **verify** (PK, input, proof)

hash = **proof2hash**(proof)

Else INVALID

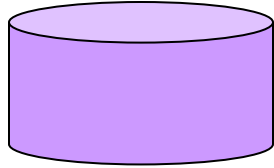
Like a regular hash function (eg SHA256)

Like a deterministic digital signature

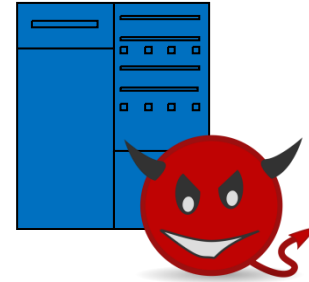
**Collision resistance. (As with SHA-256!)**

# VRF security: trusted collision resistance

Verifier **PK**



Hasher **SK**



input



proof



proof = **prove**(**SK**, input)

Like a deterministic digital signature

If **verify** (**PK**, input, proof)

hash = **proof2hash**(proof)

Else INVALID

Like a regular hash function (eg SHA256)

## Collision resistance. (As with SHA-256!)

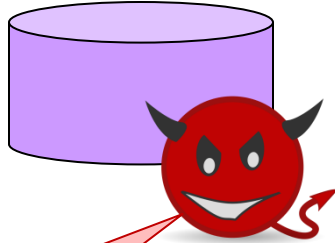
### Trusted collision resistance:

Suppose the VRF keys (**PK,SK**) are generated in a trusted way.

- If **PK** is fixed, then even an adversary that knows **SK** can't find
- ...two distinct inputs that have the same valid VRF hash

# VRF security: pseudorandomness

Verifier **PK**



I have no idea what input this hash corresponds to.

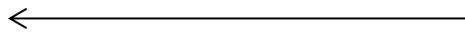
Hasher **SK**



proof = **prove** (**SK**, input)

hash = **proof2hash**(proof)

hash

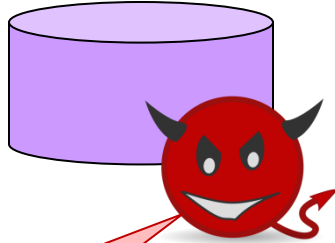


**Only the Hasher can compute the hash. (No dictionary attacks!)**



# VRF security: pseudorandomness

Verifier **PK**



I have no idea what input this hash corresponds to.

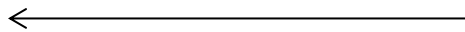
Hasher **SK**



proof = **prove** (**SK**, input)

hash = **proof2hash**(proof)

hash



**Only the Hasher can compute the hash. (No dictionary attacks!)**

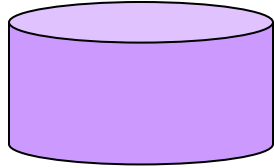
## Pseudorandomness:

Suppose the VRF keys (**PK,SK**) are generated in a trusted way.

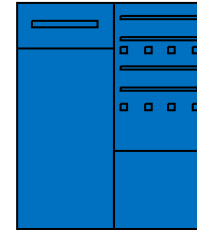
- Given an input, its VRF hash output looks pseudorandom
- ... to any adversary that does not know its proof or **SK**.

# VRFs stop dictionary attacks on hash-based structures

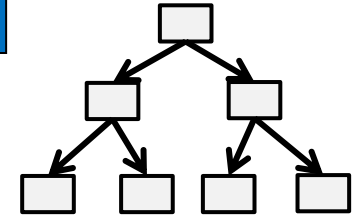
Verifier **PK**



Hasher **SK**



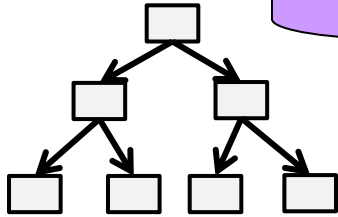
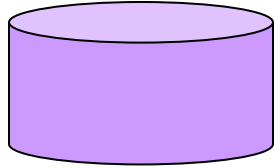
Build out of  
VRF hashes



Hash-based data  
structure

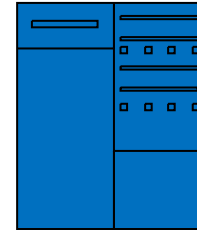
# VRFs stop dictionary attacks on hash-based structures

Verifier **PK**

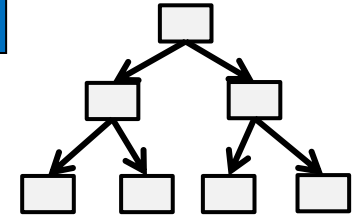


Hash-based data structure

Hasher **SK**



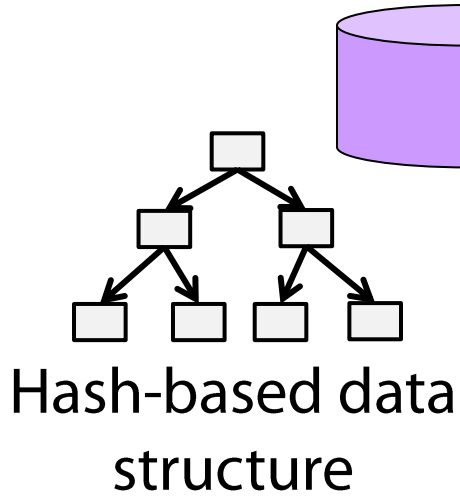
Build out of VRF hashes



Hash-based data structure

# VRFs stop dictionary attacks on hash-based structures

Verifier **PK**

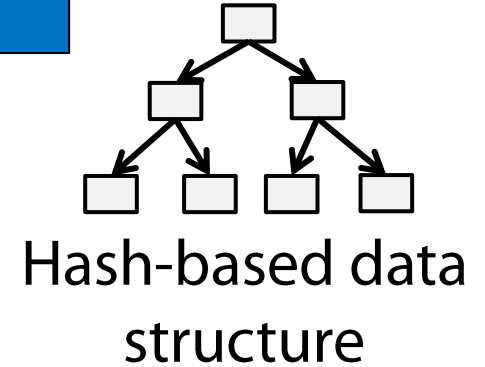


I can't use dictionary attacks to learn what is stored in this data structure.

Hasher **SK**

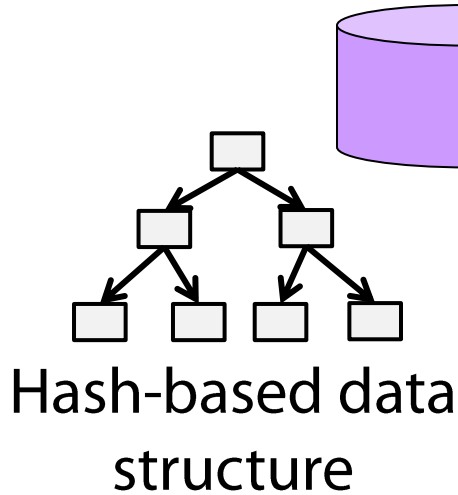


Build out of VRF hashes



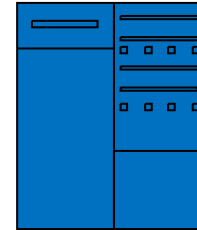
# VRFs stop dictionary attacks on hash-based structures

Verifier **PK**

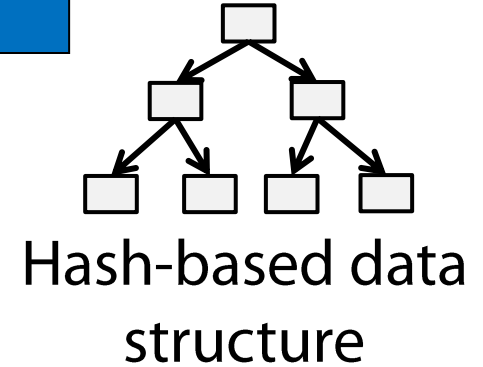


I can't use dictionary attacks to learn what is stored in this data structure.

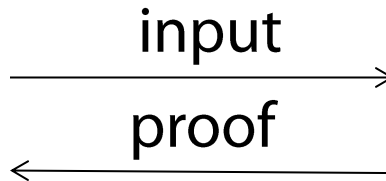
Hasher **SK**



Build out of VRF hashes



Is input in the data structure?



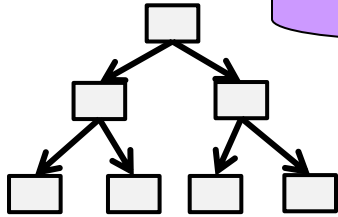
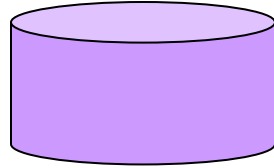
proof = **prove** (**SK**, input)

If **verify** (**PK**, input, proof)

hash = **proof2hash**(proof)

# VRFs stop dictionary attacks on hash-based structures

Verifier **PK**

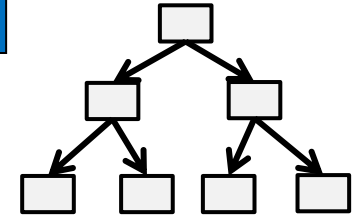


Hash-based data structure

Hasher **SK**



Build out of VRF hashes



Hash-based data structure

Is input in the data structure?

input

proof

proof = **prove** (**SK**, input)

If **verify** (**PK**, input, proof)

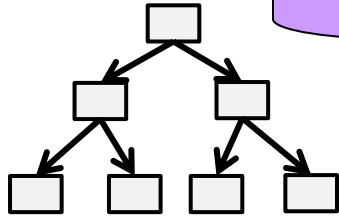
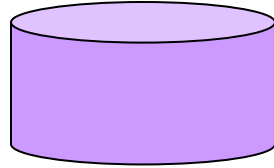
hash = **proof2hash**(proof)

Is hash in data structure?

Else INVALID

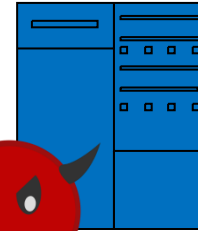
# VRFs stop dictionary attacks on hash-based structures

Verifier **PK**

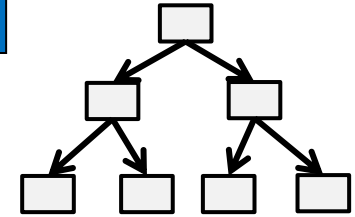


Hash-based data structure

Hasher **SK**



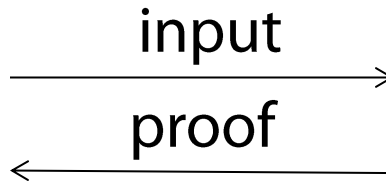
Build out of VRF hashes



Hash-based data structure

I can't lie about the hash output

Is input in the data structure?



proof = **prove** (**SK**, input)

If **verify** (**PK**, input, proof)

hash = **proof2hash**(proof)

Is hash in data structure?

Else INVALID

## **-00 draft includes**

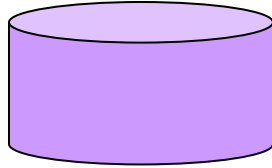
---

- VRF Security Definitions and Security Considerations
- Elliptic Curve VRF (**EC-VRF**)
  - Works with any cyclic group **G** of prime order **q** with generator **g**
  - Ciphersuites for NIST P-256 curve and Ed25519 curve
  - Algorithm is generic. Could support other curves
- RSA Full-Domain-Hash VRF (**RSA-FDH-VRF**)
- Also, we have:
  - Formal cryptographic security proofs: **<http://ia.cr/2017/099>**
  - Implementations: **<https://github.com/fcelda/nsec5-crypto>**

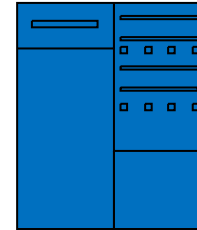


# RSA-FDH-VRF (RSA full domain hash VRF)

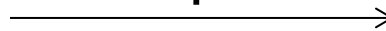
Verifier  $(N, e)$



Hasher  $(N, d)$



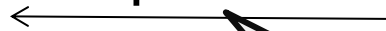
input



proof =

$$(\text{MGF1}(\text{input}))^d \bmod N$$

proof

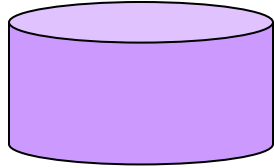


deterministic  
RSA signature

[RFC8017]

# RSA-FDH-VRF (RSA full domain hash VRF)

Verifier ( $N, e$ )



RSA signature verification

Hasher ( $N, d$ )



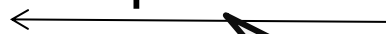
input



proof =

$$(\text{MGF1}(\text{input}))^d \bmod N$$

proof



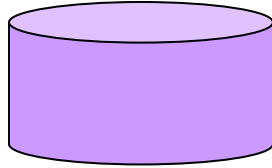
deterministic  
RSA signature

[RFC8017]

$$\text{If } \text{MGF1}(\text{input}) = (\text{proof})^d \bmod N$$

# RSA-FDH-VRF (RSA full domain hash VRF)

Verifier ( $N, e$ )



Hasher ( $N, d$ )



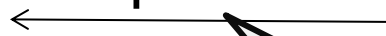
input



proof =

$$(\text{MGF1}(\text{input}))^d \bmod N$$

proof



RSA signature verification

If  $\text{MGF1}(\text{input}) = (\text{proof})^d \bmod N$

hash =  $\mathbf{H}(\text{proof})$

Else INVALID

deterministic RSA signature

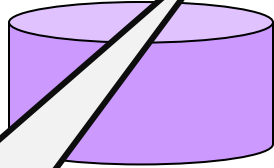
[RFC8017]

regular hash function (eg SHA256)

# EC-VRF (elliptic curve VRF)

---

Verifier  $g^x$



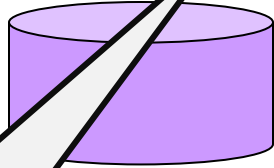
Hasher  $x$



Cyclic group  $G$  of  
prime order  $q$   
with generator  $g$   
(eg Ed25519)

# EC-VRF (elliptic curve VRF)

Verifier  $g^x$



Hasher  $x$



input



Cyclic group  $G$  of  
prime order  $q$   
with generator  $g$   
(eg Ed25519)

regular hash function  
(eg SHA256)

$h = \text{hash\_to\_curve}(\text{input})$

$y = h^x$

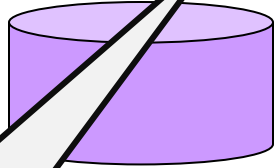
choose random nonce  $k$

$c = H(g, g^x, h, h^x, g^k, h^k)$

$s = k - cx \text{ mod } q$

# EC-VRF (elliptic curve VRF)

Verifier  $g^x$



Hasher  $x$



input



Cyclic group  $G$  of  
prime order  $q$   
with generator  $g$   
(eg Ed25519)

regular hash function  
(eg SHA256)

$h = \text{hash\_to\_curve}(\text{input})$

$y = h^x$

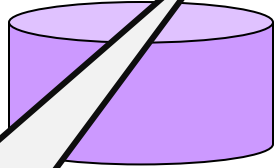
choose random nonce  $k$

$c = H(g, g^x, h, h^x, g^k, h^k)$

proof:  $(y, c, s)$   $s = k - cx \pmod q$

# EC-VRF (elliptic curve VRF)

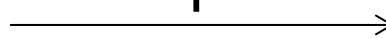
Verifier  $g^x$



Hasher  $x$



input



Cyclic group  $\mathbf{G}$  of  
prime order  $\mathbf{q}$   
with generator  $\mathbf{g}$   
(eg Ed25519)

regular hash function  
(eg SHA256)

$\mathbf{h} = \text{hash\_to\_curve}(\text{input})$

$\boldsymbol{\gamma} = \mathbf{h}^x$

choose random nonce  $\mathbf{k}$

$\mathbf{c} = \mathbf{H}(\mathbf{g}, \mathbf{g}^x, \mathbf{h}, \boldsymbol{\gamma}, \mathbf{g}^{\mathbf{k}}, \mathbf{h}^{\mathbf{k}})$

proof:  $(\boldsymbol{\gamma}, \mathbf{c}, \mathbf{s})$   $\mathbf{s} = \mathbf{k} - \mathbf{c}x \text{ mod } \mathbf{q}$

$\mathbf{u} = (\mathbf{g}^x)^{\mathbf{c}} \mathbf{g}^{\mathbf{s}}$

$\mathbf{h} = \text{hash\_to\_curve}(\text{input})$

$\mathbf{v} = \boldsymbol{\gamma}^{\mathbf{c}} \mathbf{h}^{\mathbf{s}}$

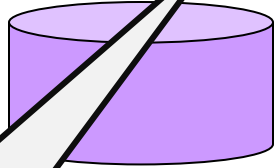
If  $\mathbf{c} = \mathbf{H}(\mathbf{g}, \mathbf{g}^x, \mathbf{h}, \boldsymbol{\gamma}, \mathbf{u}, \mathbf{v})$

hash = x-coordinate of  $\boldsymbol{\gamma}$

Else INVALID

# EC-VRF (elliptic curve VRF)

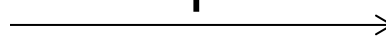
Verifier  $g^x$



Hasher  $x$



input



Cyclic group  $G$  of  
prime order  $q$   
with generator  $g$   
(eg Ed25519)

regular hash function  
(eg SHA256)

$h = \text{hash\_to\_curve}(\text{input})$

$\gamma = h^x$

choose random nonce  $k$

$c = H(g, g^x, h, h^x, g^k, h^k)$

proof:  $(\gamma, c, s)$   $s = k - cx \pmod q$

$u = (g^x)^c g^s$

$h = \text{hash\_to\_curve}(\text{input})$

$v = \gamma^c h^s$

If  $c = H(g, g^x, h, \gamma, u, v)$

hash = x-coordinate of  $\gamma$

Else INVALID

## ciphersuites

- NIST P-256 curve with SHA256
- Ed25519 curve with SHA256
- Could add other curves (eg Ed448)