RSA VRF

Andrew Huang, Chang Rey Tang

University of California, Santa Barbara

andrewhuang@ucsb.edu, tang.changrey@gmail.com

June 12, 2018

Overview

Introduction

Verifiable Random Function (VRF)

- Essentially a public-key version of a keyed cryptographic hash
- Secret key holder computes hash
- Anyone with public key can verify correctness of the hash

Motivation

Can be used to provide privacy against offline enumeration (eg. dictionary attacks) in data stored in a hash-based data structure

Other use cases:

- NSEC5 [1]
- Cryptocurrencies [2]

Basic Overview - Terminology

- SK: The private key for the VRF, generated outside of VRF
- PK: The public key for the VRF, generated outside of VRF
- α: The input to be hashed by the VRF.
- π: 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.

Basic Overview - Methods

VRF_prove(SK, α) $\rightarrow \pi$

- Invoked by the Prover, using the private key SK to construct a proof π

VRF_verify(PK, α, π) $\rightarrow$ VALID or INVALID

- Invoked by the Verifier to verify the correctness of the proof π using the public key

Properties

- Trusted Uniqueness
  - Assuming *PK* and *SK* were generated in a trustworthy manner, a computationally-bounded adversary cannot choose a VRF public key, a VRF input α, two different VRF hash outputs and two proofs π_1 and π_2, such that π_1 and π_2 are valid proofs
- Trusted collision resistance
  - Assuming *PK* and *SK* were generated in a trustworthy manner, it is computationally infeasible for an adversary to find two distinct VRF inputs that have the same VRF hash, even if that adversary knows the secret VRF key
- Full Pseudorandomness
  - If an adversarial Verifier sees a VRF hash input or output at any time without its corresponding VRF proof, then the hash output is indistinguishable from a random value.

Contribution

Implement RSA VRF in Python2 such that we achieve the following properties:

- Trusted Uniqueness
- Trusted Collision Resistance
- Full Pseudo-randomness

Note

Implementation uses RSA to generate public-private key primitives and follows the standard RSA assumption in the random oracle model. Our Python2 implementation uses the pyca/cryptography library [3] to achieve this.

Implementation

VRF_prove(SK, α, k)

- The VRF computes a proof π of length k as a deterministic RSA signature on input α using the RSA algorithm parameterized with the selected hash algorithm

VRF_verify(PK, α, π, k)

- A RSA signature verification is used to verify the correctness of the proof generated by the VRF_prove function

Parameters

RSA Public Key, $< n, e >$

- $n \rightarrow$ modulus for both public and private keys
- $e \rightarrow$ public key exponent

RSA Private Key, $< n, d >$

- $n \rightarrow$ modulus for both public and private keys
- $d \rightarrow$ private key exponent

k, length in octets of the RSA modulus n

Primitives

I2OSP

- Conversion of a non-negative integer to an octet string

OS2IP

- Conversion of an octet string to a non-negative integer

RSASP1

- RSA signature primitive

RSAVP1

- RSA verification primitive

MGF1

- Mask Generation Function based on your hash function of choice (i.e. SHA-256)

I2OSP

I2OSP converts a non-negative integer to an octet string of a specified length.

Input

$x \rightarrow$ non-negative integer to be converted
$xLen \rightarrow$ intended length of the resulting octet string

Output

$X \rightarrow$ corresponding octet string of length $xLen$

OS2IP

OS2IP converts an octet string to a non-negative integer.

Input

$X \rightarrow$ octet string to be converted

Output

$x \rightarrow$ corresponding nonnegative integer

RSASP1

RSASP1 is a standard algorithm for producing signatures using the RSA private key

Input

$< n, d > \rightarrow$ RSA private key

$m \rightarrow$ message representation, an integer between 0 and $n - 1$

Output

$s \rightarrow$ signature representation, an integer between 0 and $n - 1$

RSAVP1

RSAVP1 is a standard algorithm for verifying signatures using the RSA public key

Input

$< n, e > \rightarrow$ RSA public key

$s \rightarrow$ signature representation, an integer between 0 and $n - 1$

Output

$m \rightarrow$ message representation, an integer between 0 and $n - 1$

MGF1

MGF1 is a mask generation function based on a hash function.

Options

hash $\rightarrow$ hash function (*hLen* denotes the length in octets of the hash function output)

Input

mgfSeed $\rightarrow$ seed from which mask is generated, an octet string
maskLen $\rightarrow$ intended length in octets of the mask, at most 2^{32} *hLen*

Output

mask $\rightarrow$ mask, an octet string of length *maskLen*

Prover Implementation

Pseudocode: *VRF_prove*(SK, α, k)

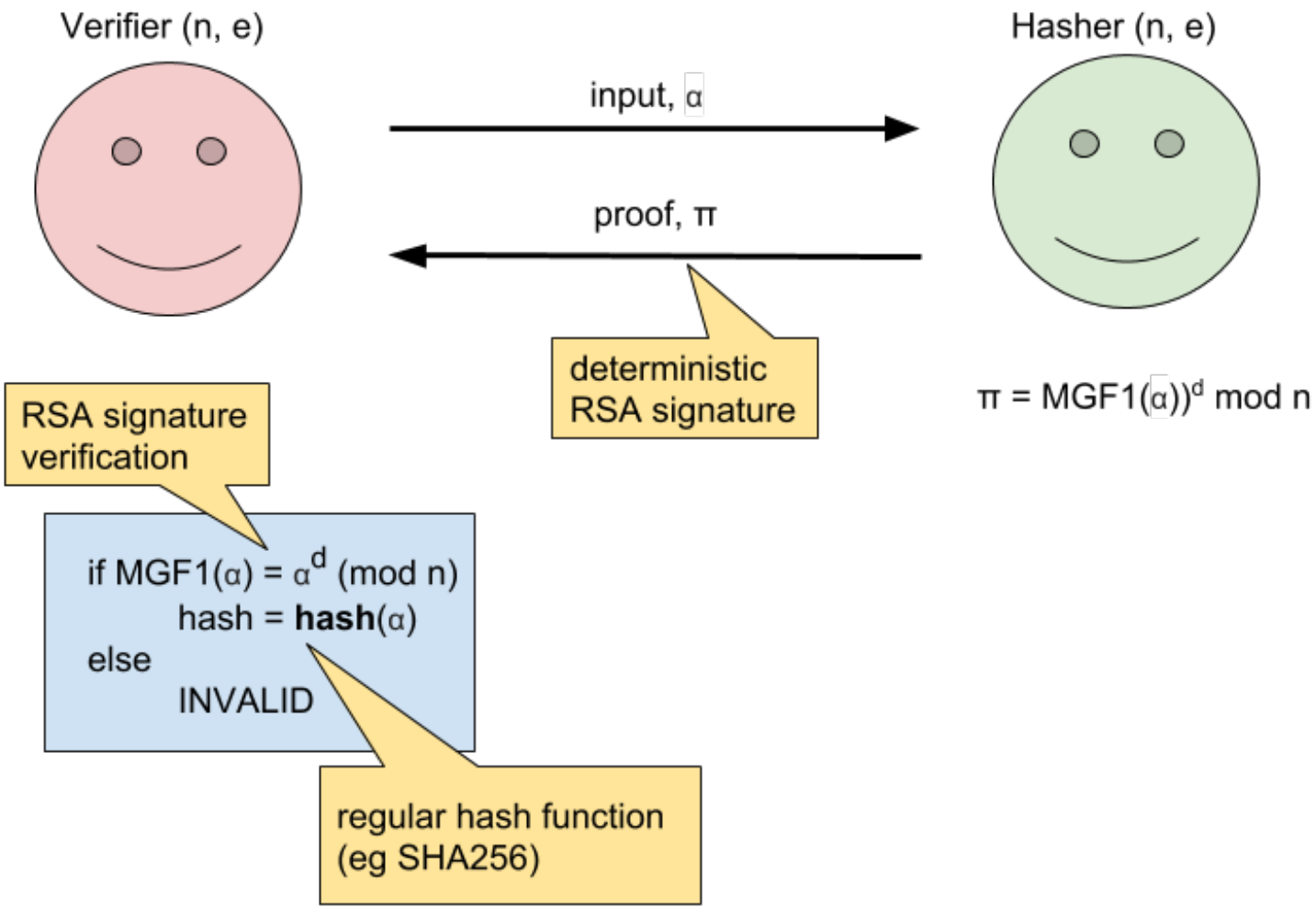
```
def VRF_prove(secret_key, alpha, k):
    EM = mgf1(alpha, k-1)
    m = os2ip(EM)
    s = secret_key.rsasp1(m)
    pi = i2osp(s, k)
    return pi
```

Verifier Implementation

Pseudocode: *VRF_verify*(PK, α, π, k)

```
def VRF_verifying(public_key, alpha, pi, k):
    s = os2ip(pi)
    m = public_key.rsavp1(s)
    EM = i2osp(m, k-1)
    EM_ = mgf1(alpha, k-1)
    if EM == EM_:
        return "VALID"
    else:
        return "INVALID"
```

RSA VRF Flow Diagram



References

NSEC5, DNSSEC Authenticated Denial of Existence
https://datatracker.ietf.org/doc/draft-vcelak-nsec5/

Y. Gilad, R. Hemo, S. Micali, G. Vlachos, N. Zeldovich
Algorand: Scaling Byzantine Agreements for Cryptocurrencies
MIT CSAIL

Pyca Cryptography Documetation
https://cryptography.io/en/latest/

Questions?