CFRG@IETF'99
Prague, July 2017

http://www.cs.bu.edu/~goldbe/projects/vrf.html

draft-goldbe-vrf-01

Verifiable Random Functions (VRF)

Sharon Goldberg (Boston University)
Dimitrios Papadopoulos (HKUST)
Jan Vcelak (NS1)

Contributor: Leonid Reyzin (Boston University)

With additional contributions from Moni Naor & Asaf Ziv (Weizmann Institute)
Shumon Huque (Salesforce), David C. Lawrence (Akamai),

hash function zoo

hash function:

- no key
- hash = **H(**input**)**
- Verify: Check hash = **H(**input**)**

SHA256

BLAKE

hash function zoo

hash function:

SHA256

BLAKE

- no key
- hash = **H(**input**)**
- Verify: Check hash = **H(**input**)**

pseudorandom function (PRF):

HMAC

- symmetric key **k**
- hash = **H(k**, input**)**
- Verify: Cannot without **k**

hash function zoo

hash function:

SHA256

BLAKE

- no key
- hash = **H(**input**)**
- Verify: Check hash = **H(**input**)**

pseudorandom function (PRF):

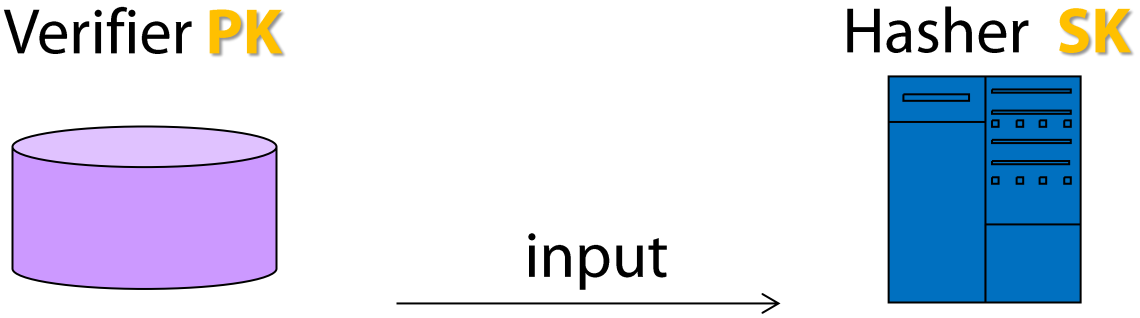
HMAC

- symmetric key **k**
- hash = **H(k**, input**)**
- Verify: Cannot without **k**

verifiable random function (VRF):

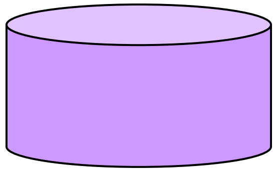
- asymmetric key **(SK**, **PK)**
- hash = **VRF_hash(SK**, input**)**
- Verify: Use **PK**

VRF: verifiable random function

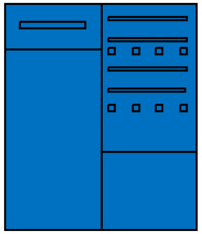


VRF: verifiable random function

Verifier **PK**



Hasher **SK**



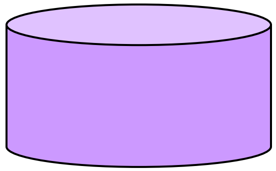
input →

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

← proof

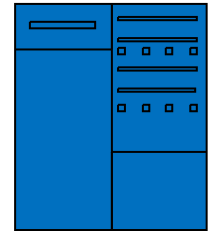
VRF: verifiable random function

Verifier **PK**



input

Hasher **SK**



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

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

hash = **proof2hash(**proof**)**

proof

Else INVALID

VRFs are useful for…

NSEC5, DNSSEC Authenticated Denial of Existence

- https://datatracker.ietf.org/doc/draft-vcelak-nsec5/
- **Our reference implementation:** https://github.com/fcelda/nsec5-crypto

CONIKS / Key Transparency / Coname / etc

- https://github.com/coniks-sys/coniks-go/blob/master/crypto/vrf/vrf.go
- https://github.com/google/keytransparency/tree/master/core/crypto/vrf
- https://github.com/yahoo/coname/tree/master/vrf

Cryptocurrencies

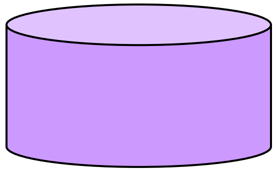
- **Algorand:** https://eprint.iacr.org/2017/454.pdf

A standard is needed.

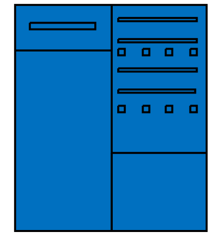
We found flaws (breaking uniqueness!) in several implementations.

VRF security: uniqueness

Verifier PK



Hasher SK



input

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

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

proof

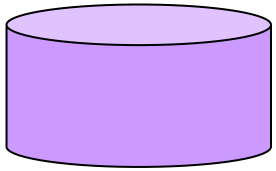
hash = **proof2hash(**proof**)**

Else INVALID

1-to-1 relationship between input & hash. (Like a hash function!)

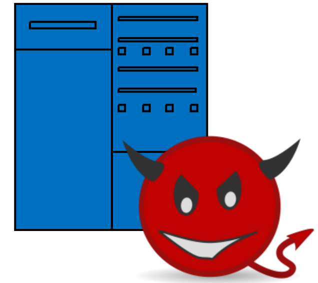
VRF security: uniqueness

Verifier PK



input →

Hasher SK



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

← proof

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

hash = **proof2hash(**proof**)**

Else INVALID

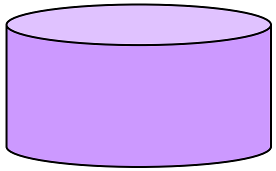
1-to-1 relationship between input & hash. (Like a hash function!)

Uniqueness:

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

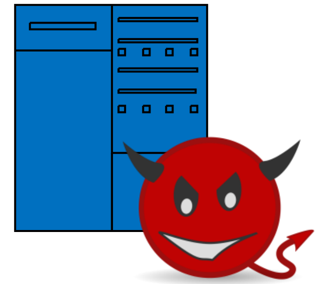
VRF security: collision resistance

Verifier PK



input

Hasher SK



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

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

hash = **proof2hash(**proof**)**

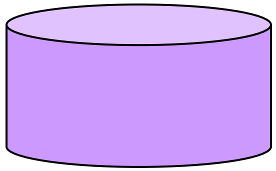
Else INVALID

proof

Collision resistance. (Like a hash function!)

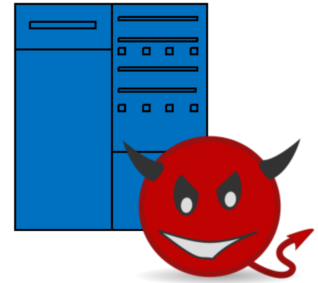
VRF security: collision resistance

Verifier **PK**



input

Hasher **SK**



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

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

hash = **proof2hash(**proof**)**

Else INVALID

proof

Collision resistance. (Like a hash function!)

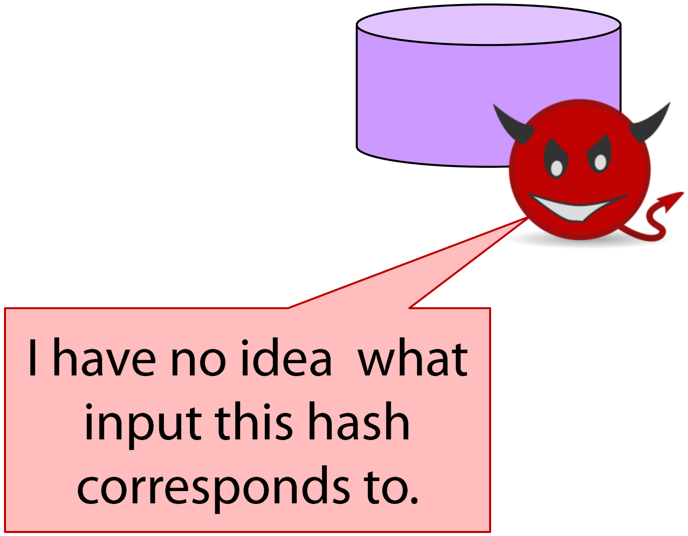
Collision resistance:

If **PK** is fixed, then even an adversary that knows **SK** can't find

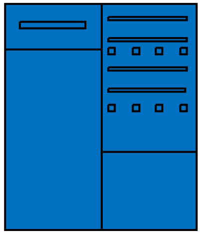
…two distinct inputs that have the same valid VRF hash

VRF security: pseudorandomness

Verifier **PK**



Hasher **SK**



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

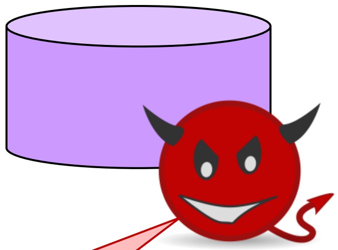
hash = **proof2hash**(proof)

hash

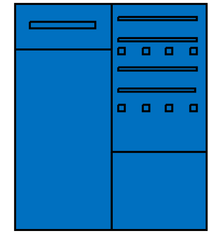
Only the Hasher can compute the hash.
(Like a PRF whose key you don't know!)

VRF security: pseudorandomness

Verifier **PK**



Hasher **SK**



I have no idea what input this hash corresponds to.

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

hash = **proof2hash**(proof)

hash ←

Only the Hasher can compute the hash.
(Like a PRF whose key you don't know!)

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 are useful for…

NSEC5, DNSSEC Authenticated Denial of Existence

- https://datatracker.ietf.org/doc/draft-vcelak-nsec5/
- **Our reference implementation:** https://github.com/fcelda/nsec5-crypto

CONIKS / Key Transparency / Coname / etc

- https://github.com/coniks-sys/coniks-go/blob/master/crypto/vrf/vrf.go
- https://github.com/google/keytransparency/tree/master/core/crypto/vrf
- https://github.com/yahoo/coname/tree/master/vrf

Preventing offline dictionary attacks on hash-based data structures

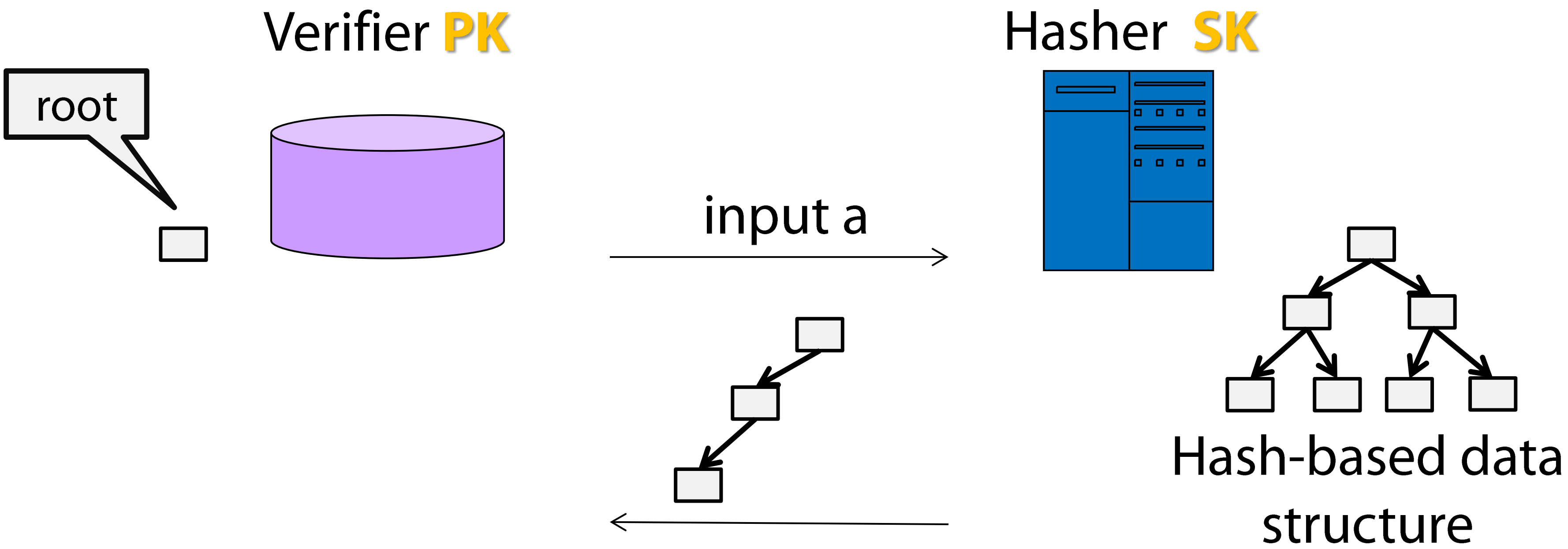
Cryptocurrencies

- **Algorand:** https://eprint.iacr.org/2017/454.pdf

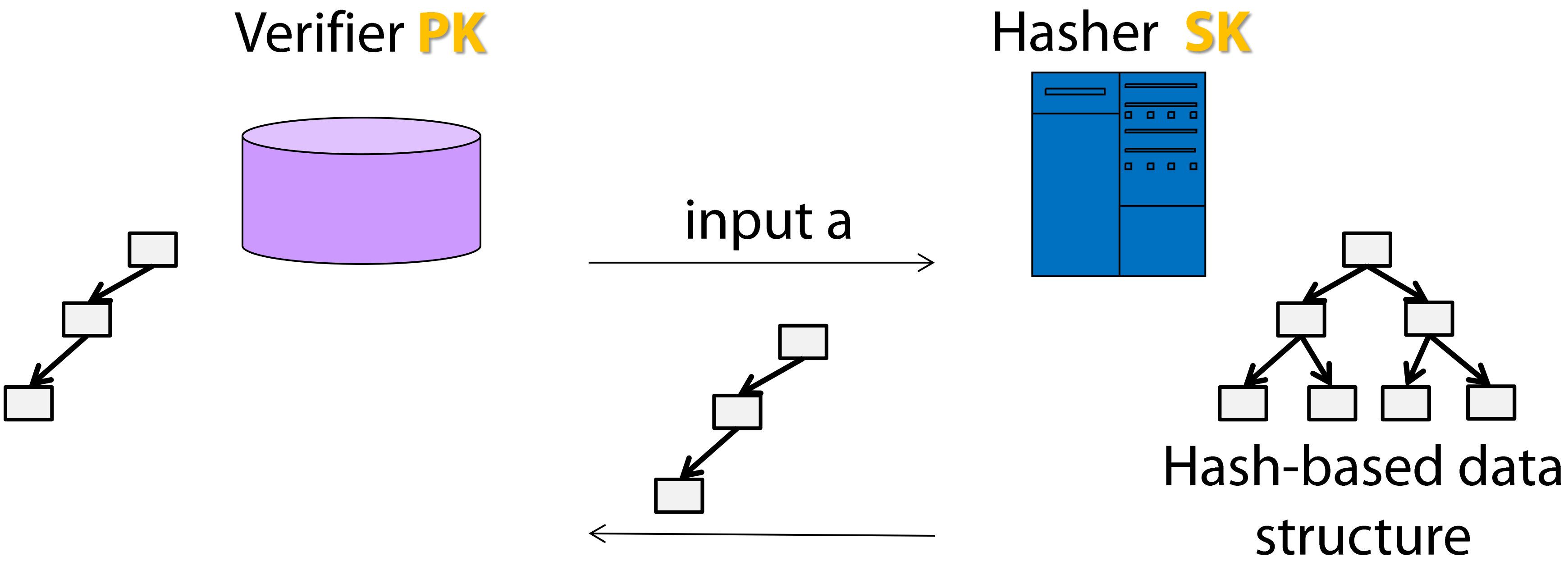
A standard is needed.

We found flaws (breaking uniqueness!) in several implementations.

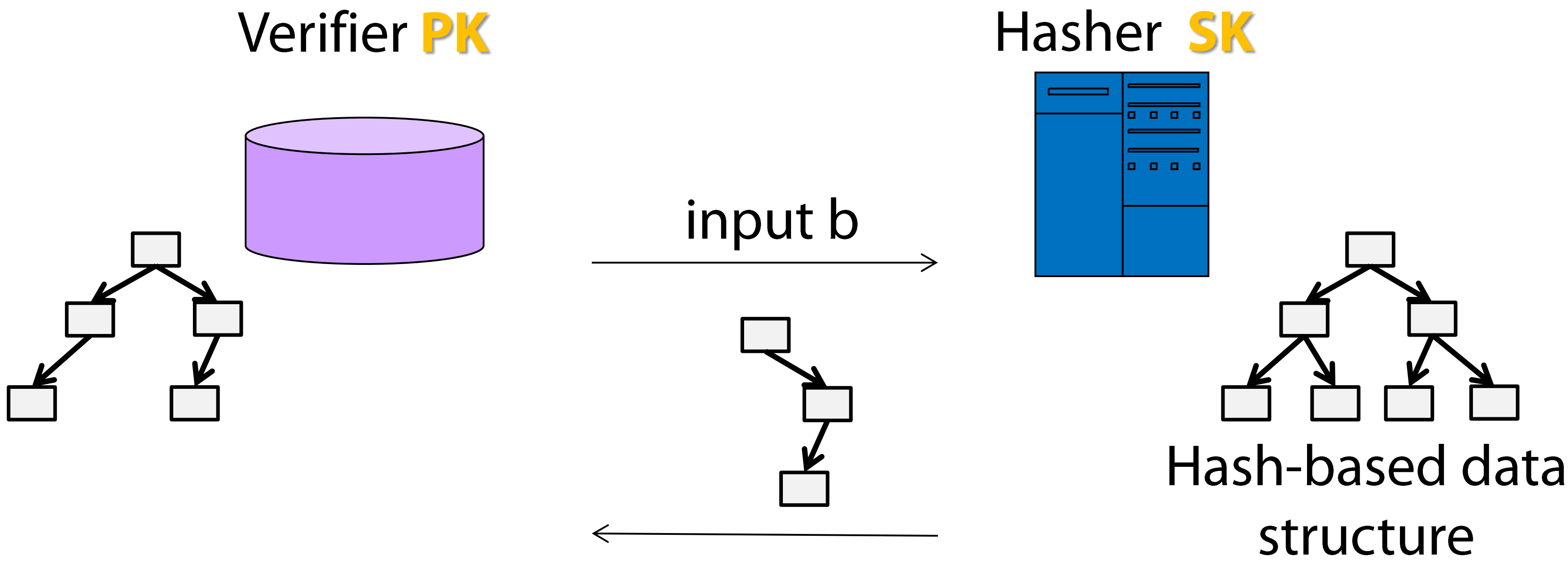
hash-based data structures



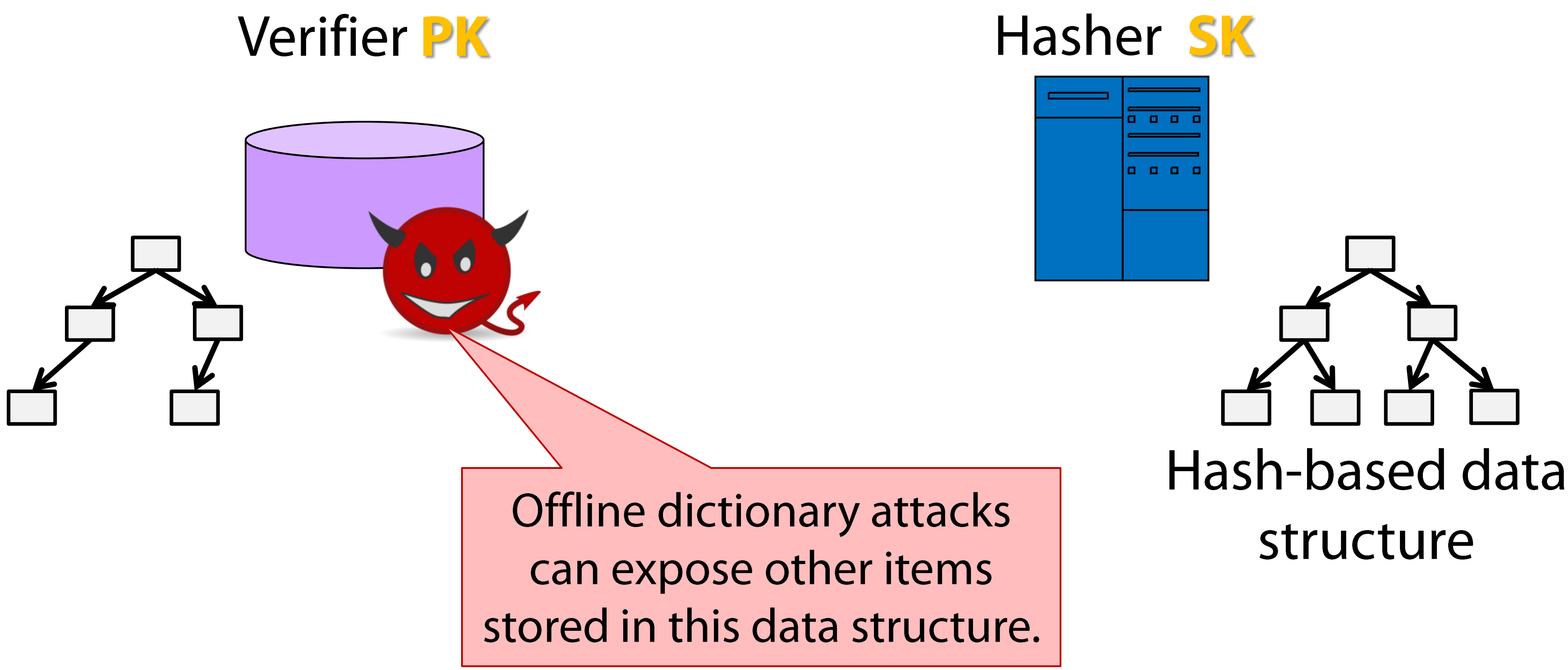
hash-based data structures



hash-based data structures

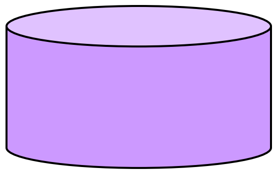


offline dictionary attacks

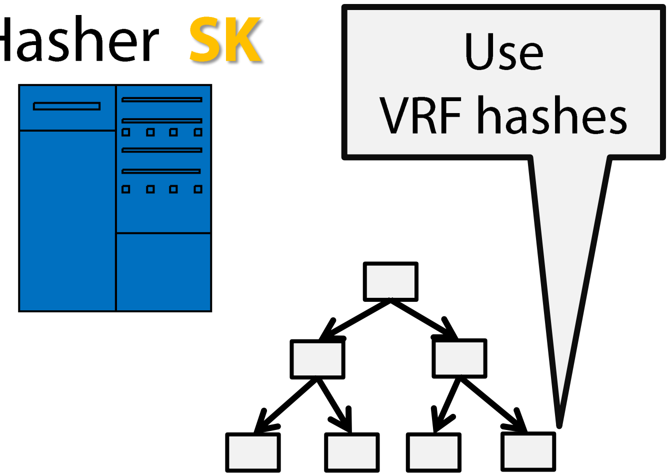


using VRFs for hash-based datastructures

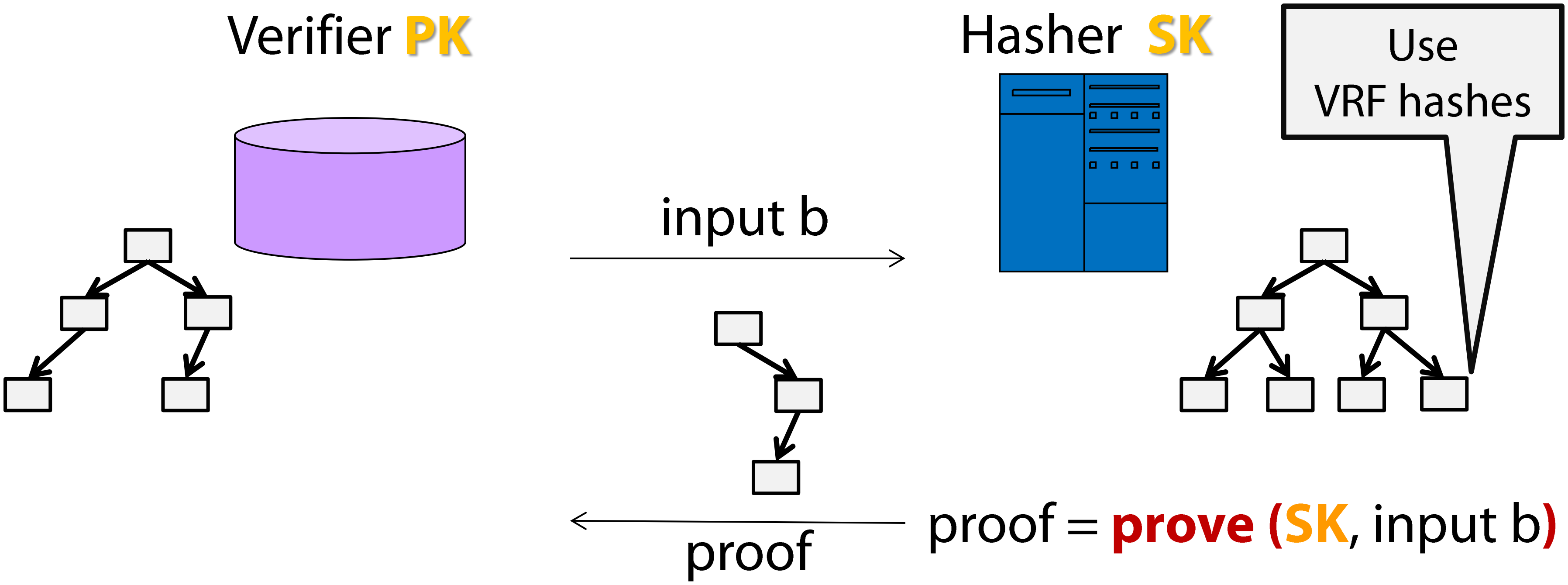
Verifier PK



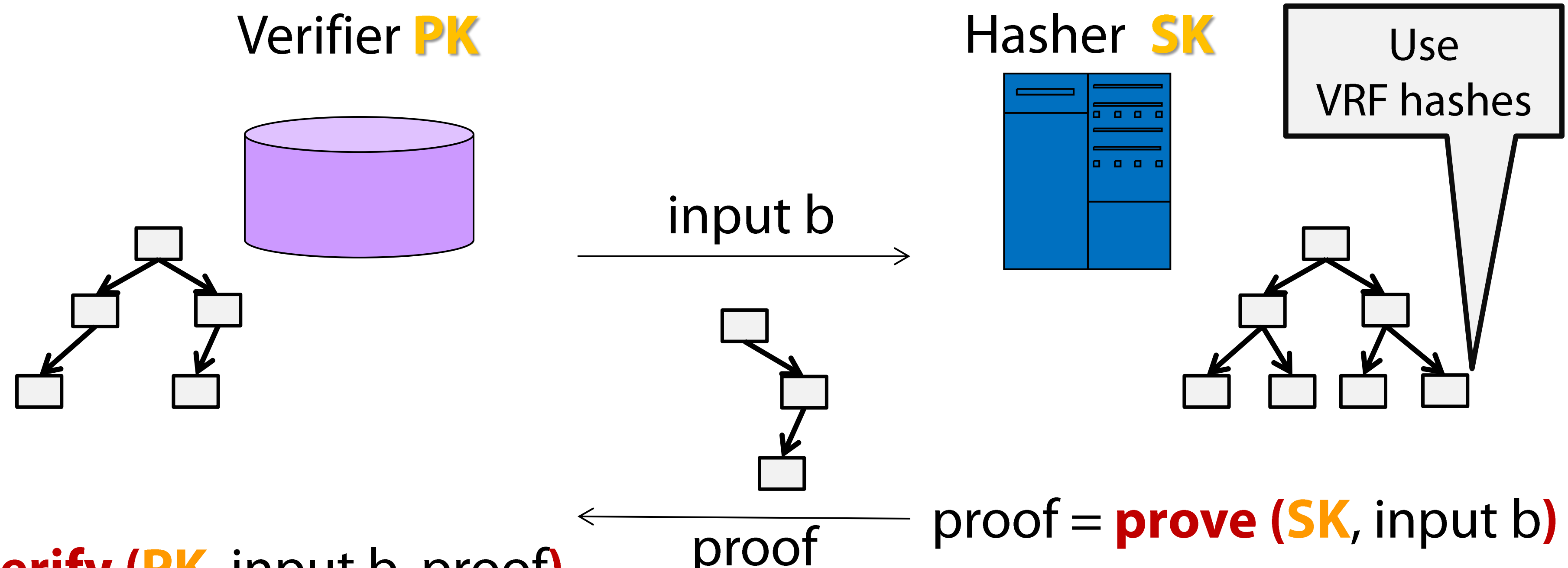
Hasher SK



using VRFs for hash-based datastructures



using VRFs for hash-based datastructures



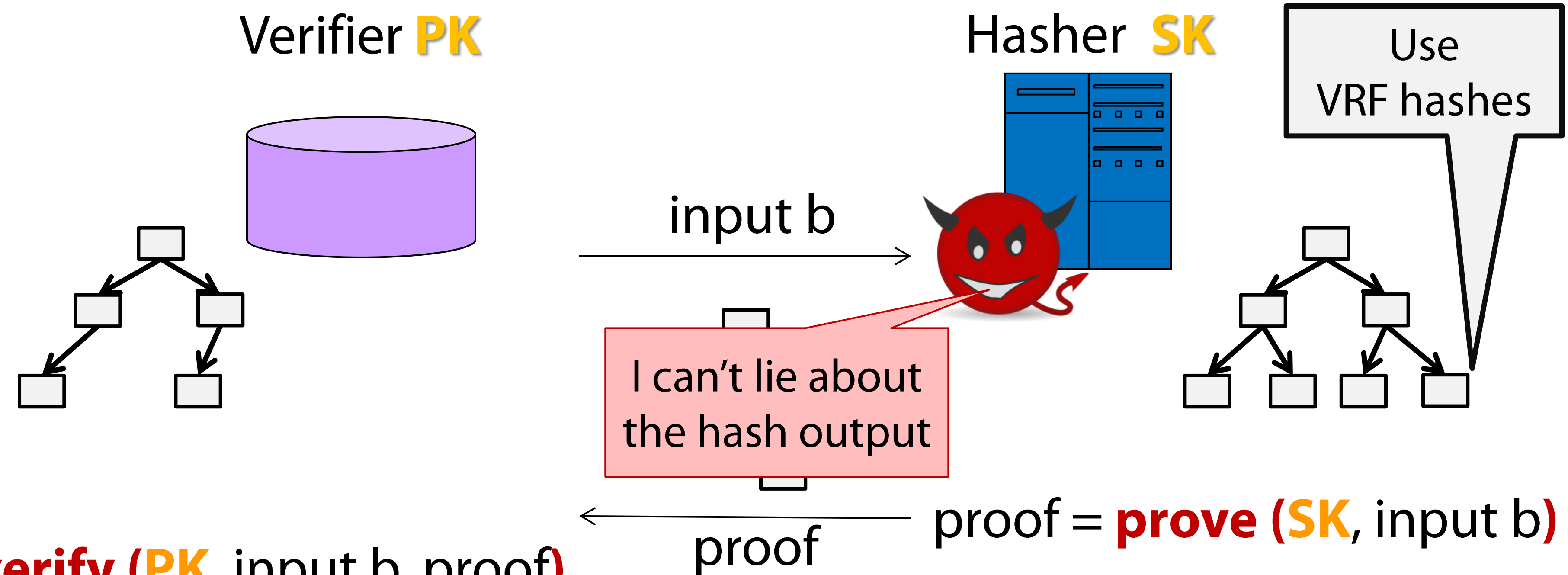
If **verify** (PK, input b, proof)

hash = **proof2hash**(proof)

hash is in data structure?

Else INVALID

using VRFs for hash-based datastructures



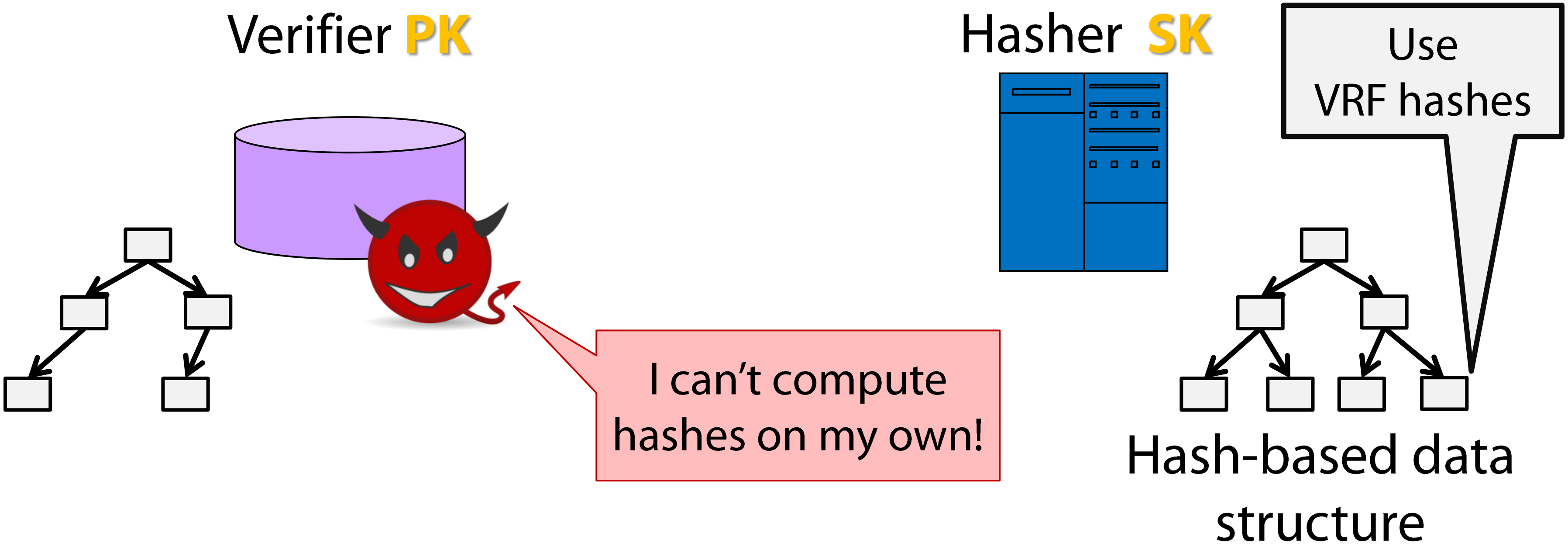
If **verify** (PK, input b, proof)

hash = **proof2hash**(proof)

hash is in data structure?

Else INVALID

VRFs stop offline dictionary attacks

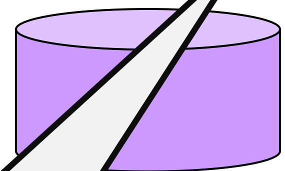


draft-goldbe-vrf-01 includes

- VRF Security Definitions and Security Considerations
- Elliptic Curve VRF (**EC-VRF**)
  - Fast. Optimized for short proofs.
  - Generic algorithm.
    - Ciphersuites for NIST P-256 curve and Ed25519 curve.
    - Could add ciphersuites for other curves like Ed448
  - **New in -01:**
    - Optimized for curves with cofactor > 1 (i.e. Ed25519)
    - Added “key validation” function, so uniqueness and collision resistance hold even if the public key is generated adversarially
- RSA Full-Domain-Hash VRF (**RSA-FDH-VRF**)
- Backed by concrete cryptographic security proofs with careful analysis, fixing bugs in prior work: http://ia.cr/2017/099

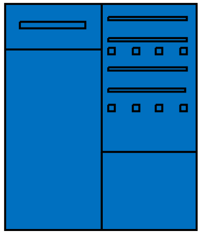
EC-VRF (elliptic curve VRF)

Verifier $PK = g^x$



Cyclic group $\mathbf{G}$ of prime order $\mathbf{q}$ with generator $\mathbf{g}$ (eg NIST P256 curve)

Hasher x



input →

$\mathbf{h}$ = hash_to_curve (input)

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

← proof: $(\boldsymbol{\gamma}, \boxed{\mathbf{c}, \mathbf{s}})$

zero-knowledge proof:
$\boldsymbol{\gamma} = \mathbf{h}^{\mathbf{x}}$ and $\mathbf{PK} = \mathbf{g}^{\mathbf{x}}$
have same discrete log

hash = $\mathbf{H}(\boldsymbol{\gamma})$

EC-VRF (elliptic curve VRF)

Verifier $PK = g^x$

Hasher x

input

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

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

choose random nonce $\mathbf{k}$

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

$\mathbf{s} = \mathbf{k - cx \bmod q}$

proof: $(\boldsymbol{\gamma}, \boxed{\mathbf{c, s}})$

zero-knowledge proof:

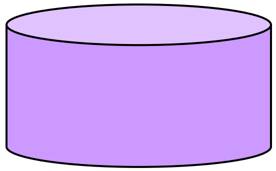
$\boldsymbol{\gamma} = \mathbf{h^x}$ and $\mathbf{PK = g^x}$

have same discrete log

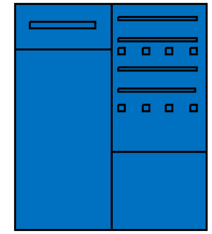
hash $= \mathbf{H(\gamma)}$

EC-VRF (elliptic curve VRF)

Verifier $PK = g^x$



Hasher x



input →

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

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

choose random nonce $\mathbf{k}$

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

$\mathbf{s} = \mathbf{k} - \mathbf{cx} \bmod \mathbf{q}$

← proof: $(\boldsymbol{\gamma}, \boxed{\mathbf{c}, \mathbf{s}})$

zero-knowledge proof:

$\boldsymbol{\gamma} = \mathbf{h}^{\mathbf{x}}$ and $\mathbf{PK} = \mathbf{g}^{\mathbf{x}}$

have same discrete log

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

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

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

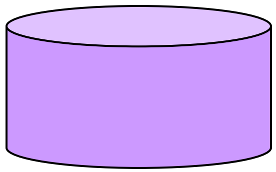
If $\mathbf{c} = \mathbf{H}(\mathbf{g}, \mathbf{PK'}, \mathbf{h}, \boldsymbol{\gamma}, \mathbf{u}, \mathbf{v})$

hash $= \mathbf{H}(\boldsymbol{\gamma})$

Else INVALID

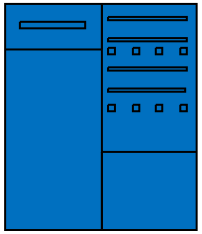
EC-VRF ciphersuites

Verifier $PK=g^x$



input →

Hasher x



$\mathbf{h}$ = hash_to_curve (input)

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

choose random nonce $\mathbf{k}$

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

$\mathbf{s} = \mathbf{k} - \mathbf{cx} \bmod \mathbf{q}$

← proof: $(\gamma, \mathbf{c}, \mathbf{s})$

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

$\mathbf{h}$ = hash_to_curve (input)

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

If $\mathbf{c} = \mathbf{H}(\mathbf{g}, \mathbf{PK}', \mathbf{h}, \gamma, \mathbf{u}, \mathbf{v})$

hash = $\mathbf{H}(\gamma)$

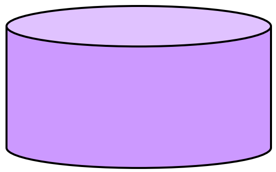
Else INVALID

ciphersuites

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

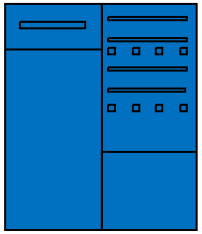
EC-VRF ciphersuites

Verifier $PK = g^x$



input

Hasher x



We spec generic hash based on SHA256. (Instead, could use Elligator with Ed25519)

$\mathbf{h}$ = hash_to_curve (input)

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

choose random nonce $\mathbf{k}$

regular hash function (eg SHA256)

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

proof: $(\boldsymbol{\gamma}, \mathbf{c}, \mathbf{s})$

$\mathbf{s} = \mathbf{k} - \mathbf{cx} \bmod \mathbf{q}$

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

$\mathbf{h}$ = hash_to_curve (input)

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

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

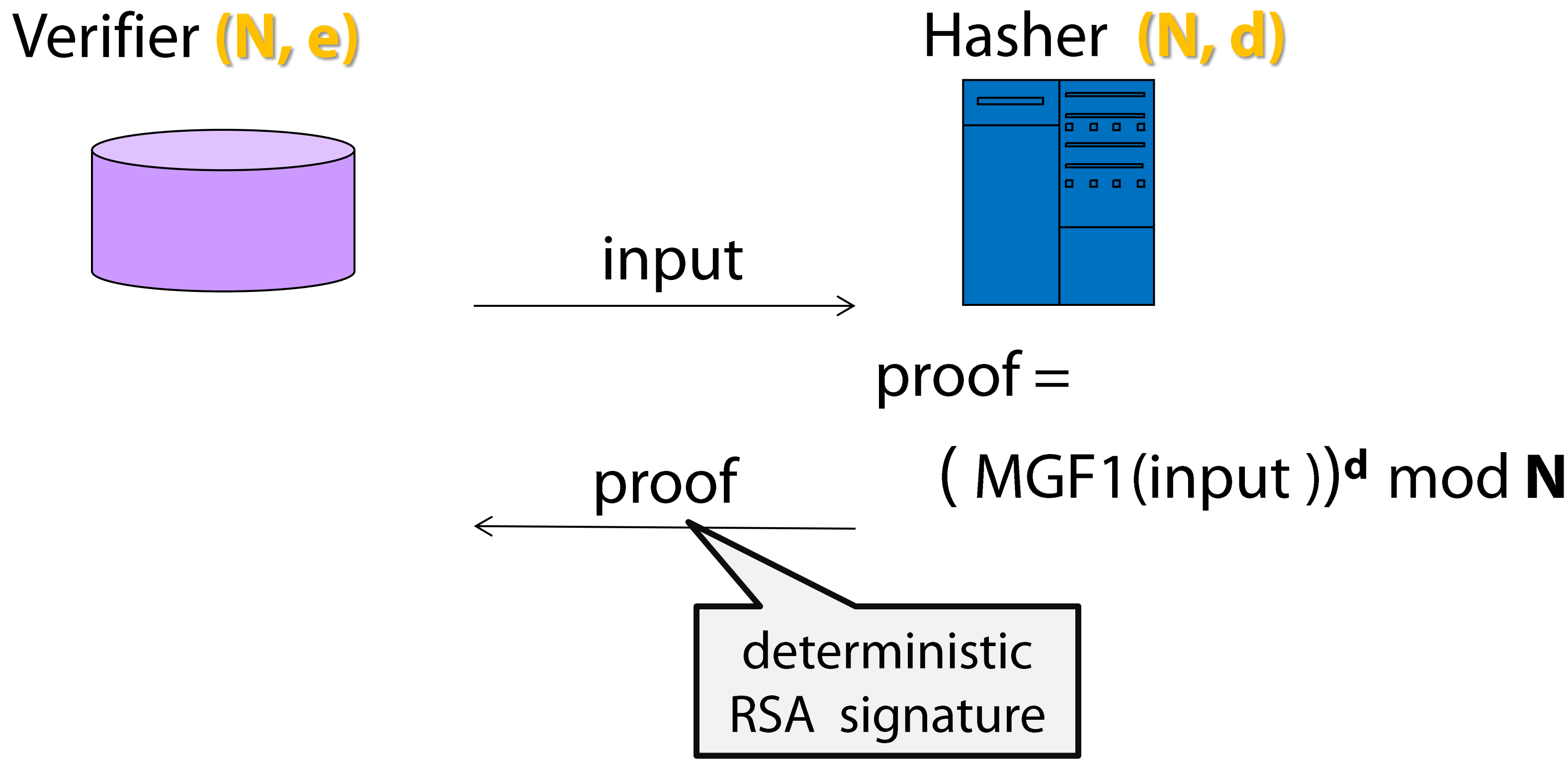
hash = $\mathbf{H}(\boldsymbol{\gamma})$

Else INVALID

ciphersuites

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

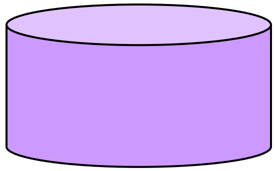
RSA-FDH-VRF (RSA full domain hash VRF)

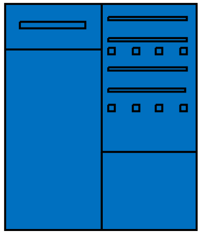


RSA-FDH-VRF (RSA full domain hash VRF)

Verifier (N, e)

Hasher (N, d)





input

proof =

$(\boxed{\mathrm{MGF1}}(\mathrm{input}))^{d} \bmod \mathbf{N}$

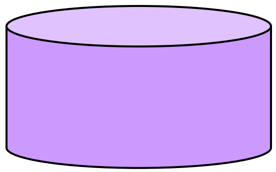
proof

deterministic RSA signature

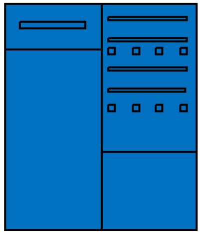
[RFC8017]

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

Verifier (N, e)



Hasher (N, d)



input →

proof =

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

[RFC8017]

← proof

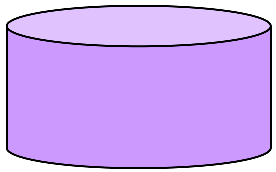
deterministic RSA signature

RSA signature verification

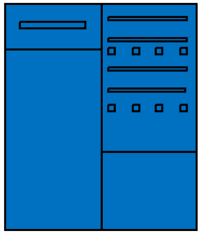
If $\text{MGF1}(\text{input}) = (\text{proof})^{d} \bmod \mathbf{N}$

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

Verifier **(N, e)**



Hasher **(N, d)**



input →

proof =

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

[RFC8017]

← proof

deterministic
RSA signature

RSA signature
verification

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

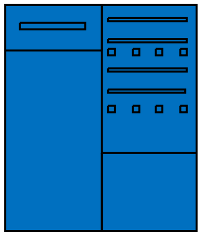
hash = **H(** proof **)**

Else INVALID

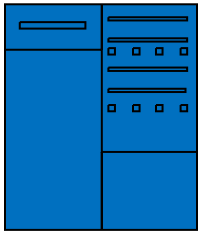
regular hash function
(eg SHA256)

Algorand cryptocurrency uses VRF for leader election

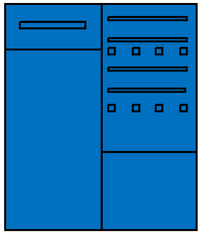
Server SK_2



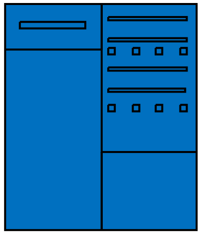
Server SK_4



Server SK_2



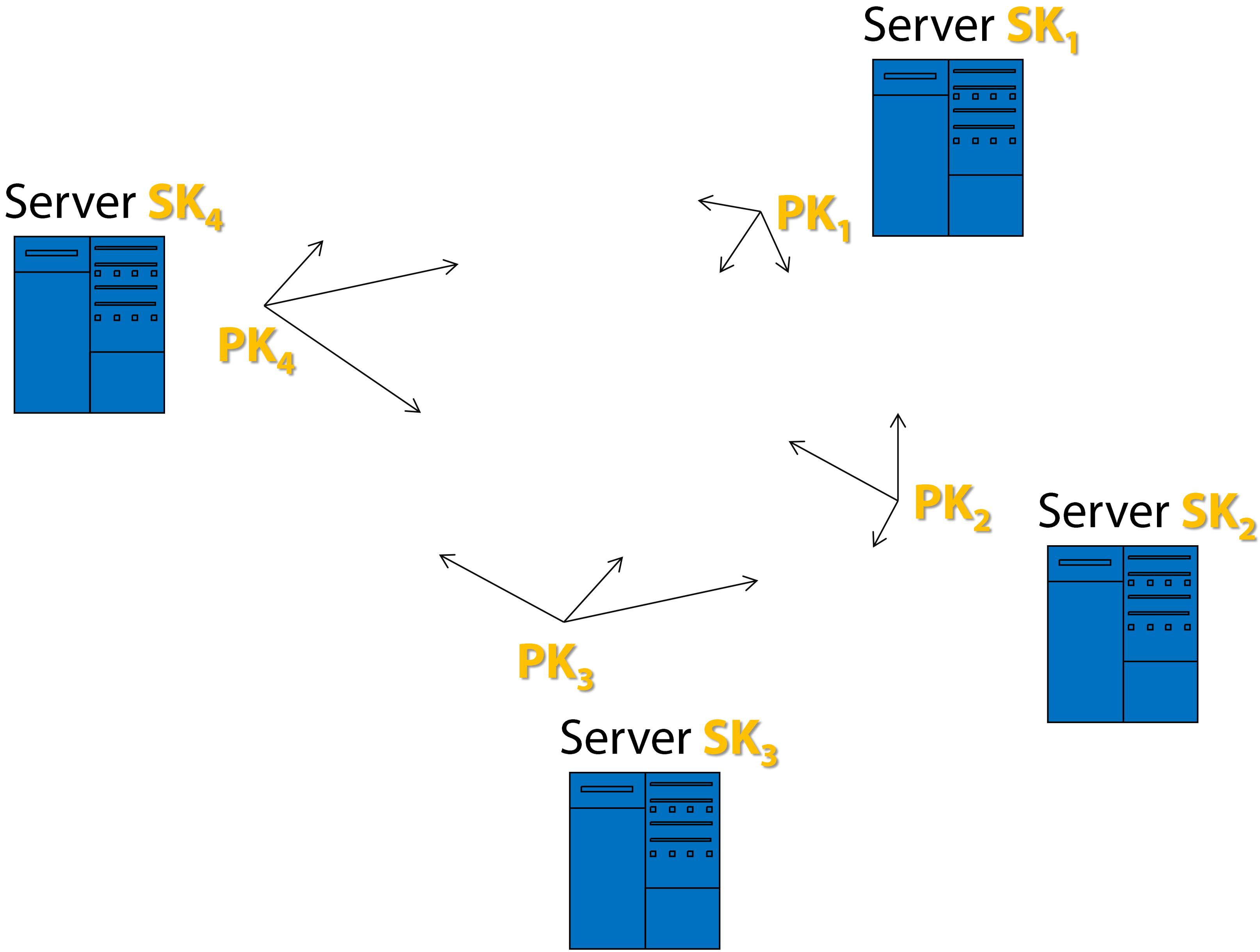
Server SK_3



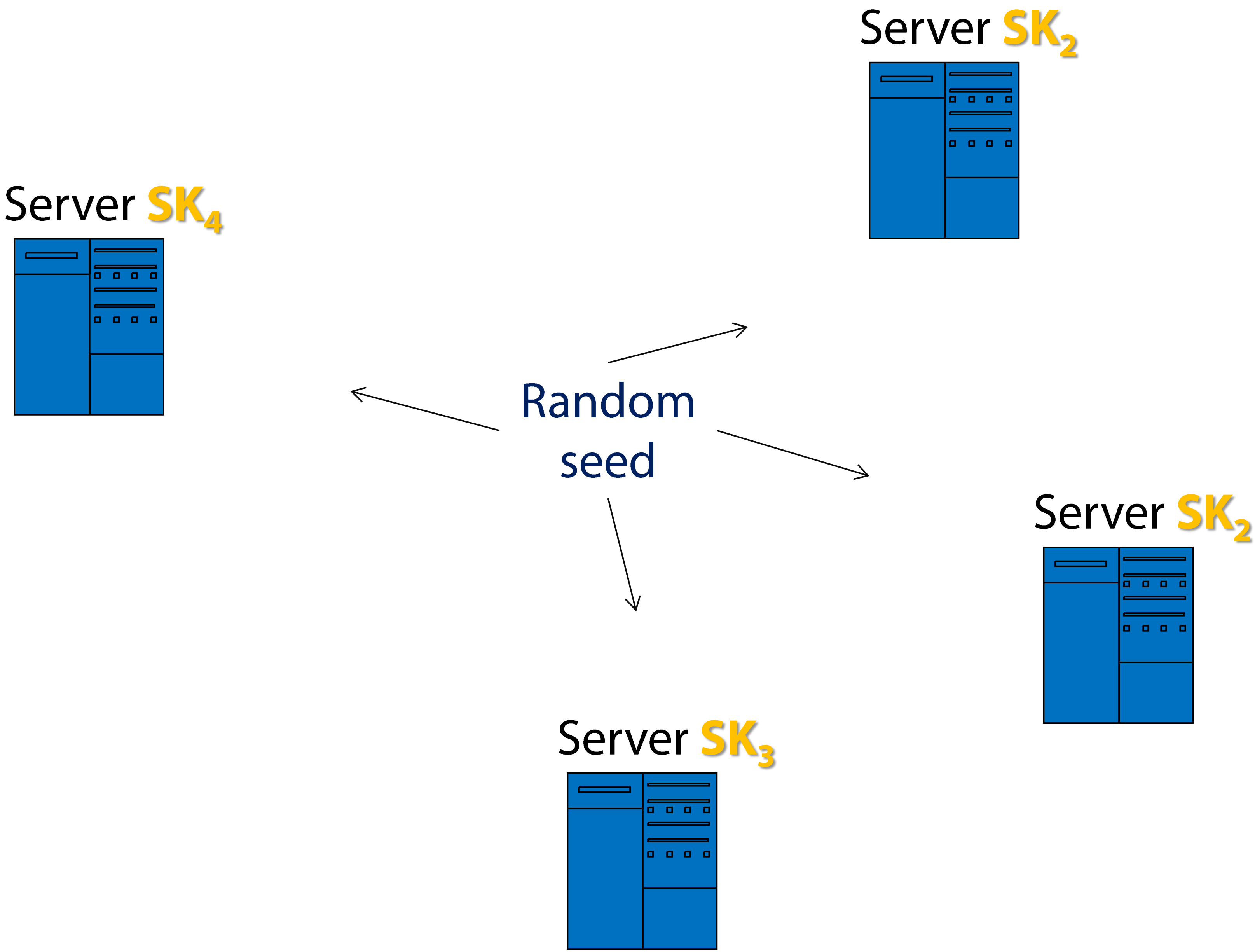
$\text{proof}_3 = \textbf{prove}\,(SK_3, \text{seed})$

$h_3 = \textbf{proof2hash}(\text{proof}_3)$

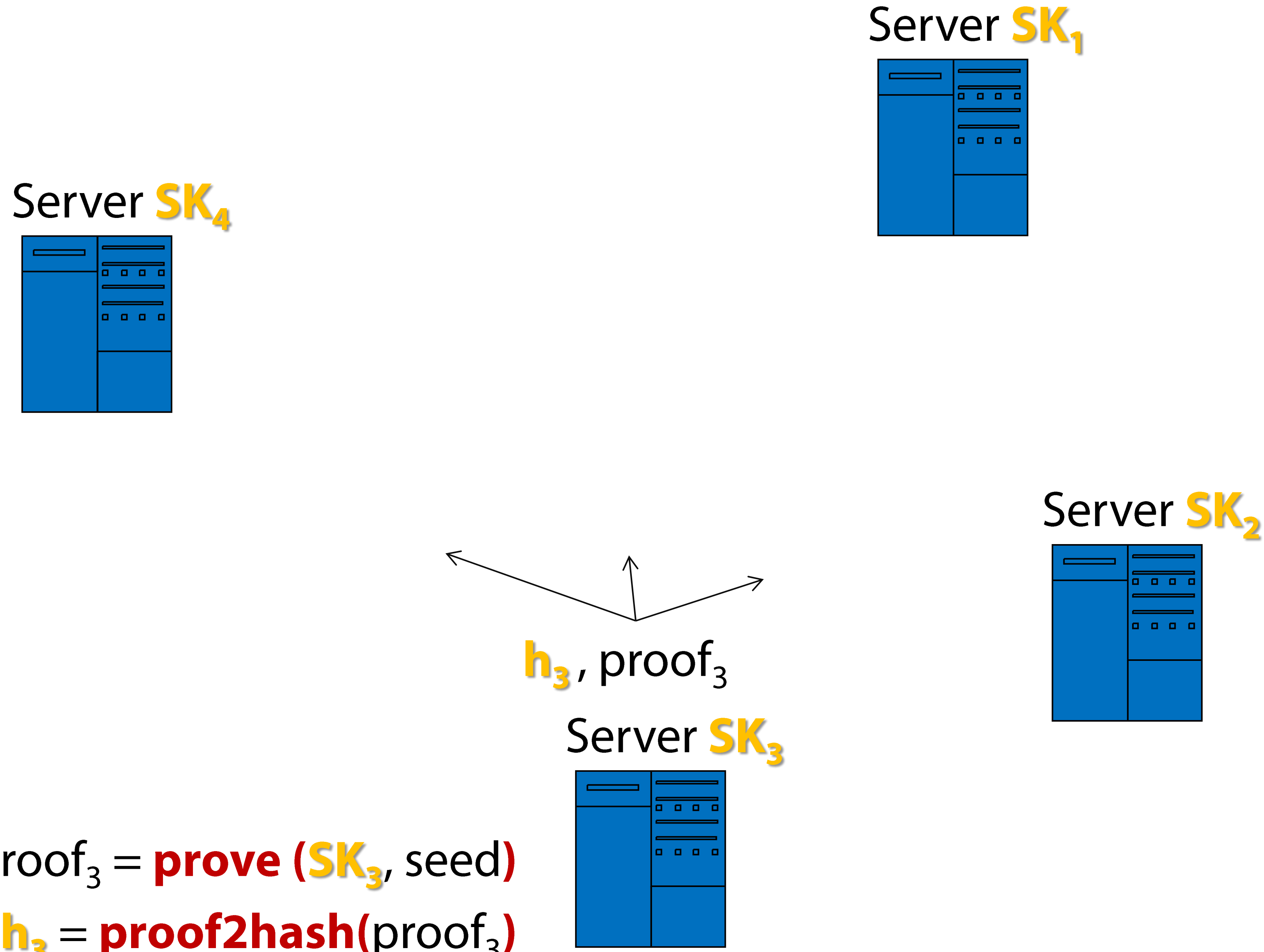
Algorand cryptocurrency uses VRF for leader election



Algorand cryptocurrency uses VRF for leader election



Algorand cryptocurrency uses **VRF** for leader election

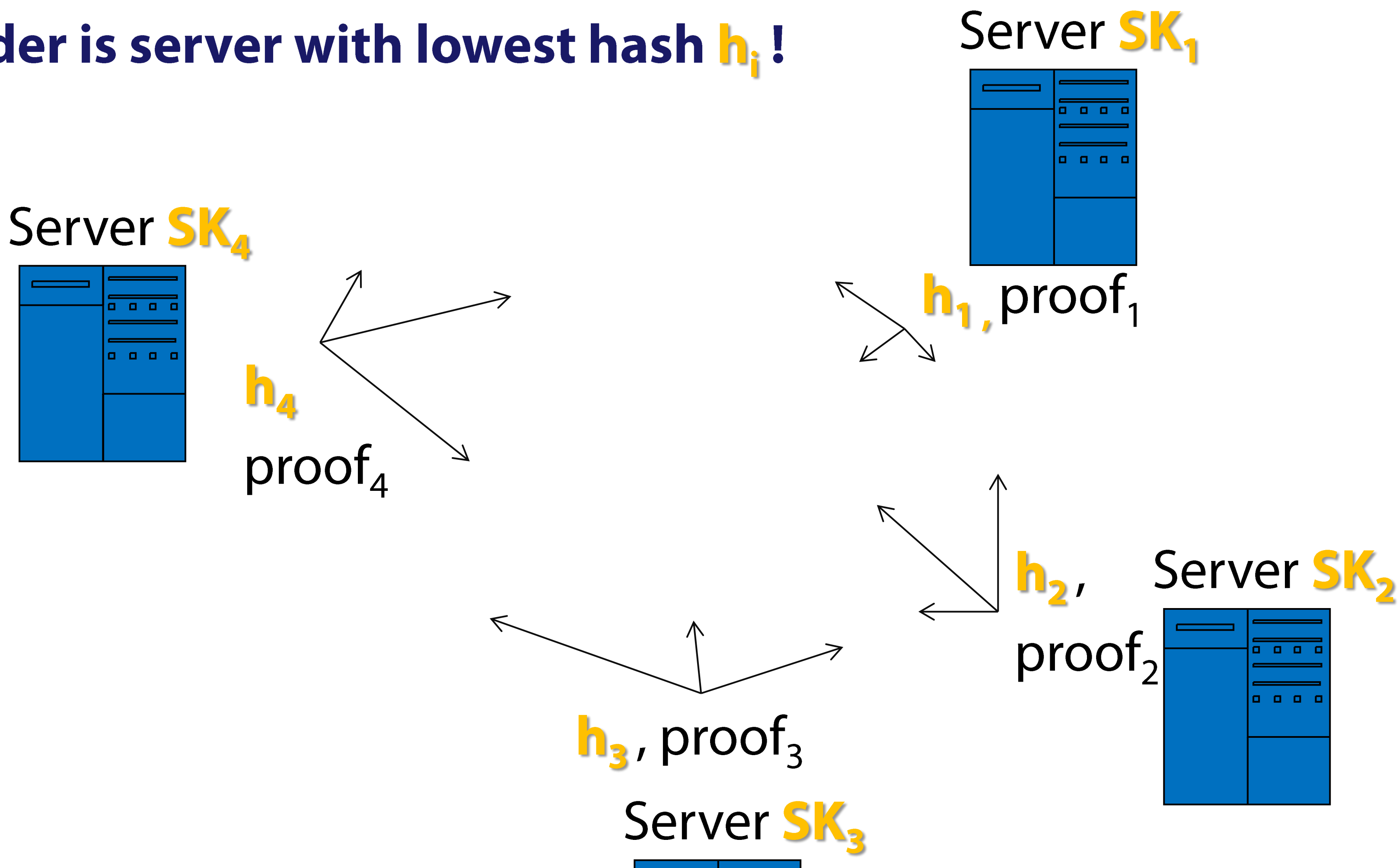


$proof_3$ = **prove** (SK_3, seed)

h_3 = **proof2hash**($proof_3$)

Algorand cryptocurrency uses VRF for leader election

Leader is server with lowest hash h_i !

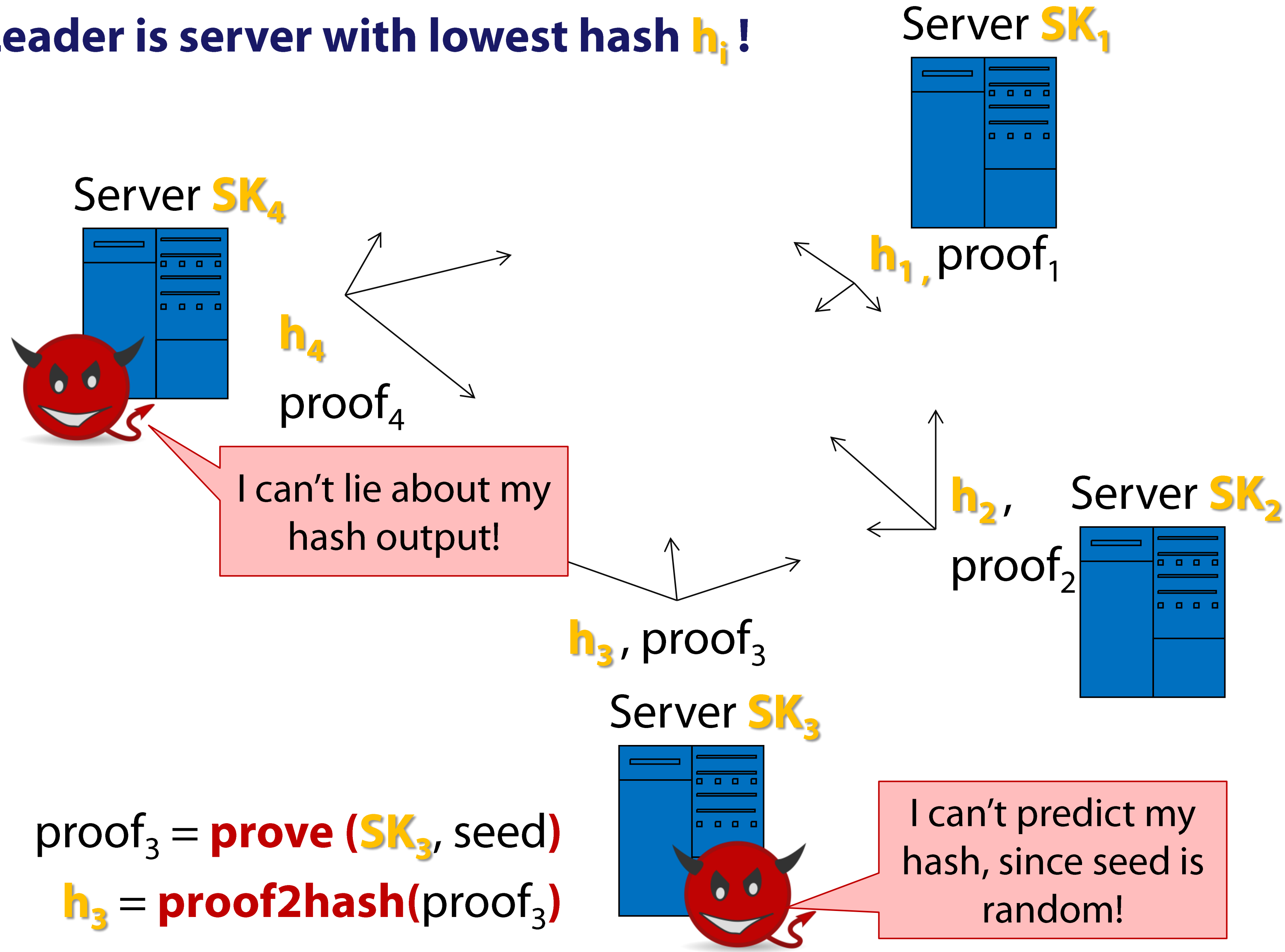


$proof_3$ = **prove** (SK_3, seed)

h_3 = **proof2hash**($proof_3$)

Algorand cryptocurrency uses VRF for leader election

Leader is server with lowest hash h_i!



$proof_3$ = **prove** (SK_3, seed)

h_3 = **proof2hash**($proof_3$)