Hash-based Signatures

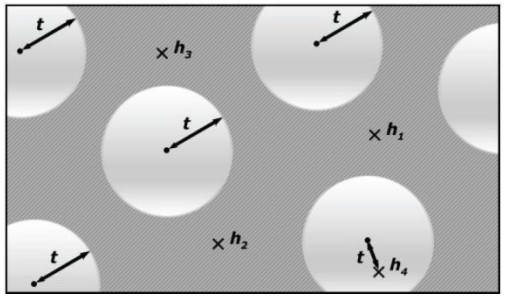
Andreas Hülsing

Summer School on Post-Quantum Cryptography
June 2017, TU Eindhoven

Post-Quantum Signatures

Lattice, MQ, Coding

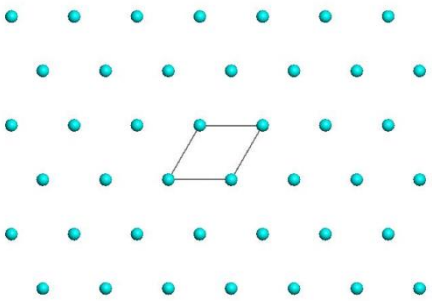
- Signature and/or key sizes
- Runtimes
- Secure parameters



$$y_1 = x_1^2 + x_1x_2 + x_1x_4 + x_3$$

$$y_2 = x_3^2 + x_2x_3 + x_2x_4 + x_1 + 1$$

$$y_3 = \ldots$$



Hash-based Signature Schemes

[Mer89]

- Post quantum
- Only secure hash function
- Security well understood
- Fast

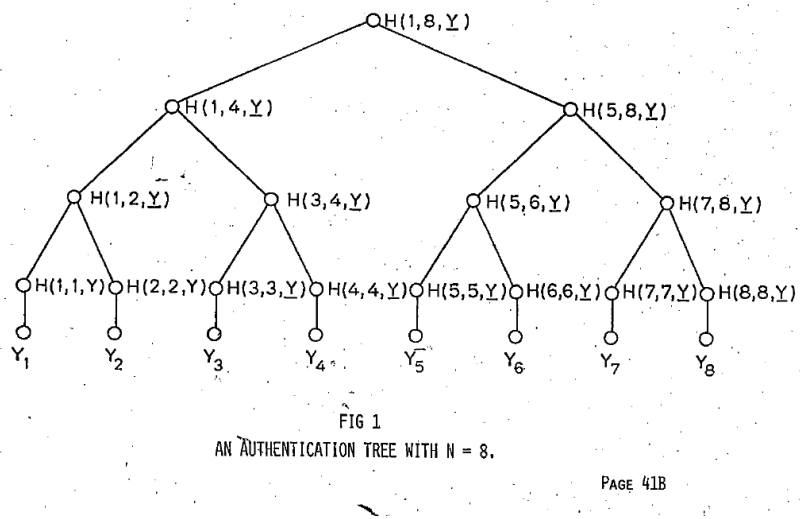
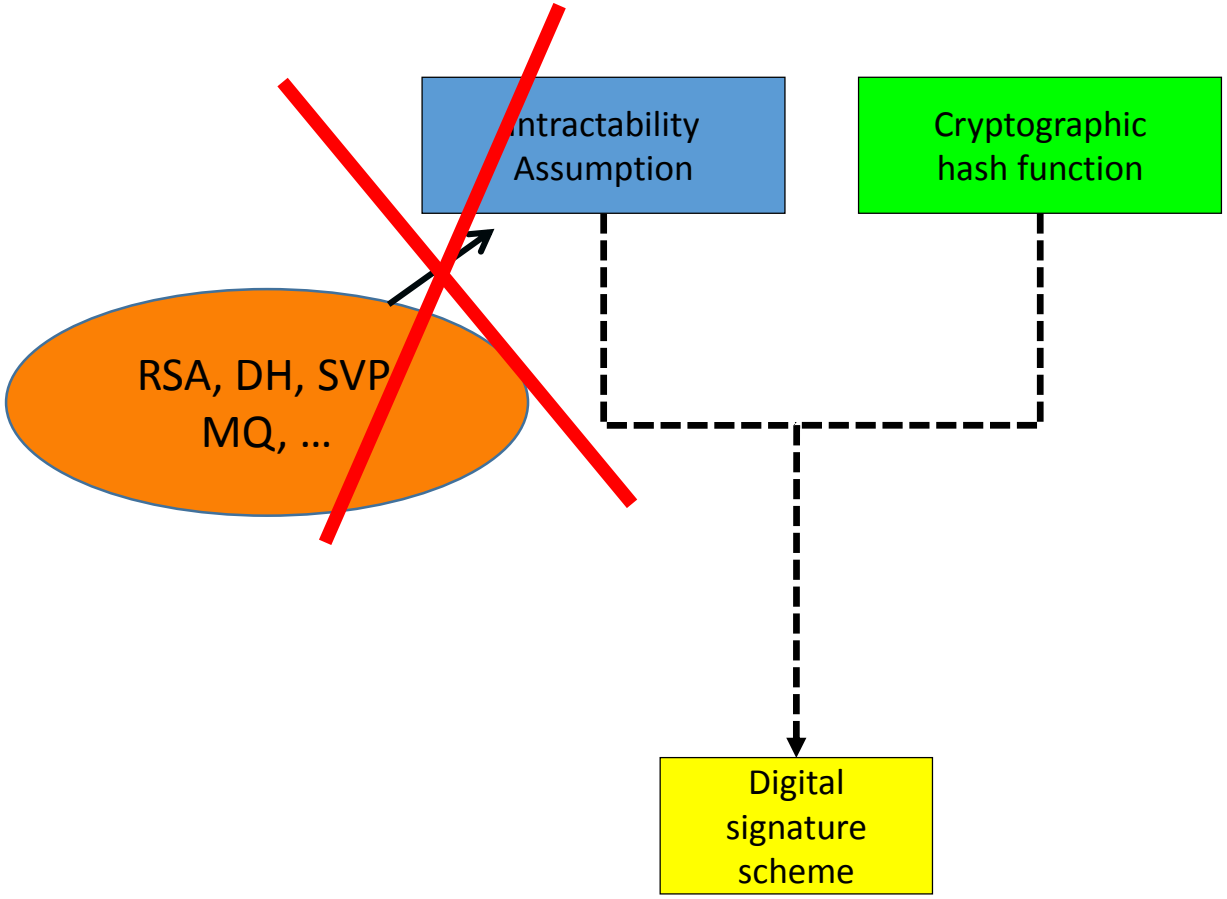


FIG 1

AN AUTHENTICATION TREE WITH N = 8.

PAGE 41B

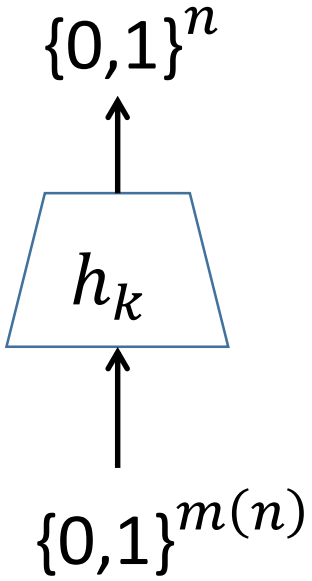
RSA – DSA – EC-DSA…



Hash function families

(Hash) function families

- $H_n := \{h_k : \{0,1\}^{m(n)} \to \{0,1\}^n\}$
- $m(n) \geq n$
- „efficient“



One-wayness

$H_n := \{h_k : \{0,1\}^{m(n)} \to \{0,1\}^n\}$

$h_k \xleftarrow{\$} H_n$

$x \xleftarrow{\$} \{0,1\}^{m(n)}$

$y_c \leftarrow h_k(x)$

Success if $h_k(x^*) = y_c$

Collision resistance

$H_n := \{h_k : \{0,1\}^{m(n)} \to \{0,1\}^n\}$

$h_k \xleftarrow{\$} H_n$

Success if

$h_k(x_1^*) = h_k(x_2^*)$ and
$x_1^* \neq x_2^*$

Second-preimage resistance

$H_n := \{h_k : \{0,1\}^{m(n)} \to \{0,1\}^n\}$

$h_k \xleftarrow{\$} H_n$

$x_c \xleftarrow{\$} \{0,1\}^{m(n)}$

Success if

$h_k(x_c) = h_k(x^*)$ and

$x_c \neq x^*$

Undetectability

$H_n := \{h_k : \{0,1\}^{m(n)} \to \{0,1\}^n\}$

$h_k \xleftarrow{\$} H_n$

$b \xleftarrow{\$} \{0,1\}$

If $b = 1$

$x \xleftarrow{\$} \{0,1\}^{m(n)}$

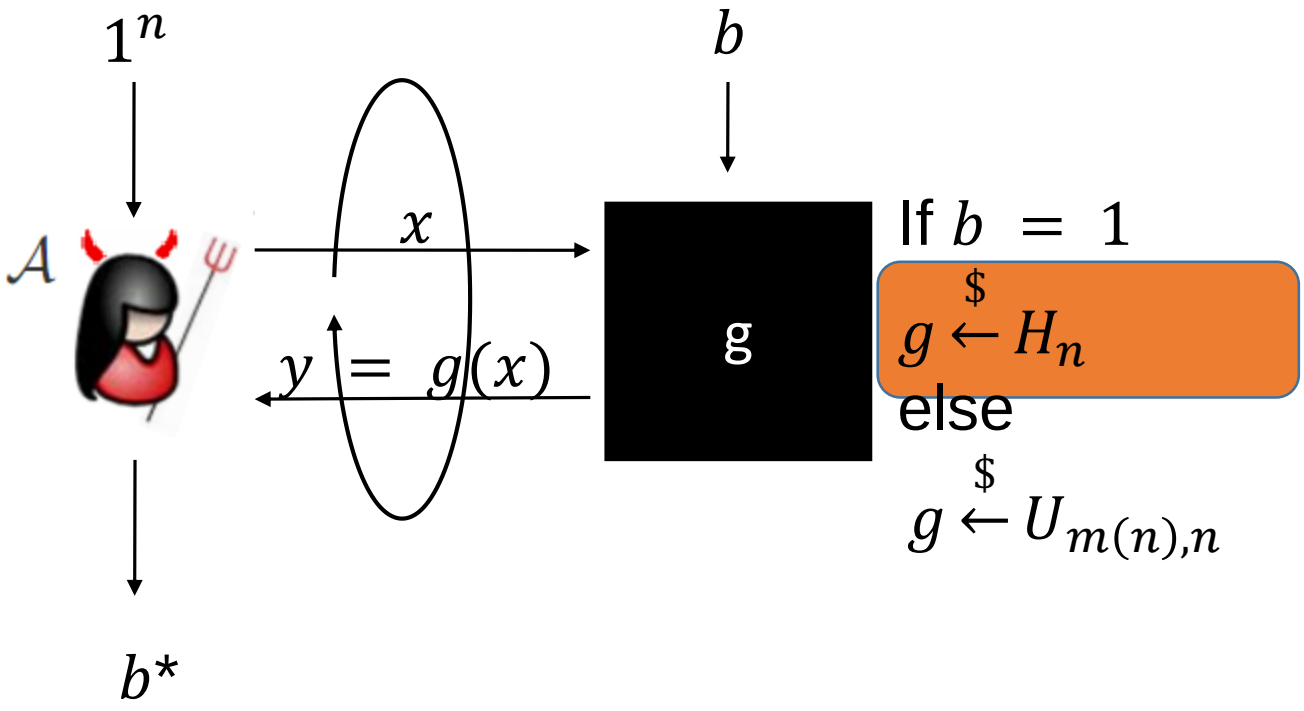
$y_c \leftarrow h_k(x)$

else

$y_c \xleftarrow{\$} \{0,1\}^n$

Pseudorandomness

$H_n := \{h_k : \{0,1\}^{m(n)} \to \{0,1\}^n\}$



Generic security

- „Black Box“ security (best we can do without looking at internals)
  - For hash functions: Security of random function family
- (Often) expressed in #queries (query complexity)
- Hash functions not meeting generic security considered insecure

Generic Security - OWF

Classically:

- No query: Output random guess

$$Succ_A^{OW} = \frac{1}{2^n}$$

- One query: Guess, check, output new guess

$$Succ_A^{OW} = \frac{2}{2^n}$$

- q-queries: Guess, check, repeat q-times, output new guess

$$Succ_A^{OW} = \frac{q+1}{2^n}$$

- Query bound: $\Theta(2^n)$

Generic Security - OWF

Quantum:

- More complex
- Reduction from quantum search for random H
- Know lower & upper bounds for quantum search!

- Query bound: $\Theta(2^{n/2})$

- Upper bound uses variant of Grover

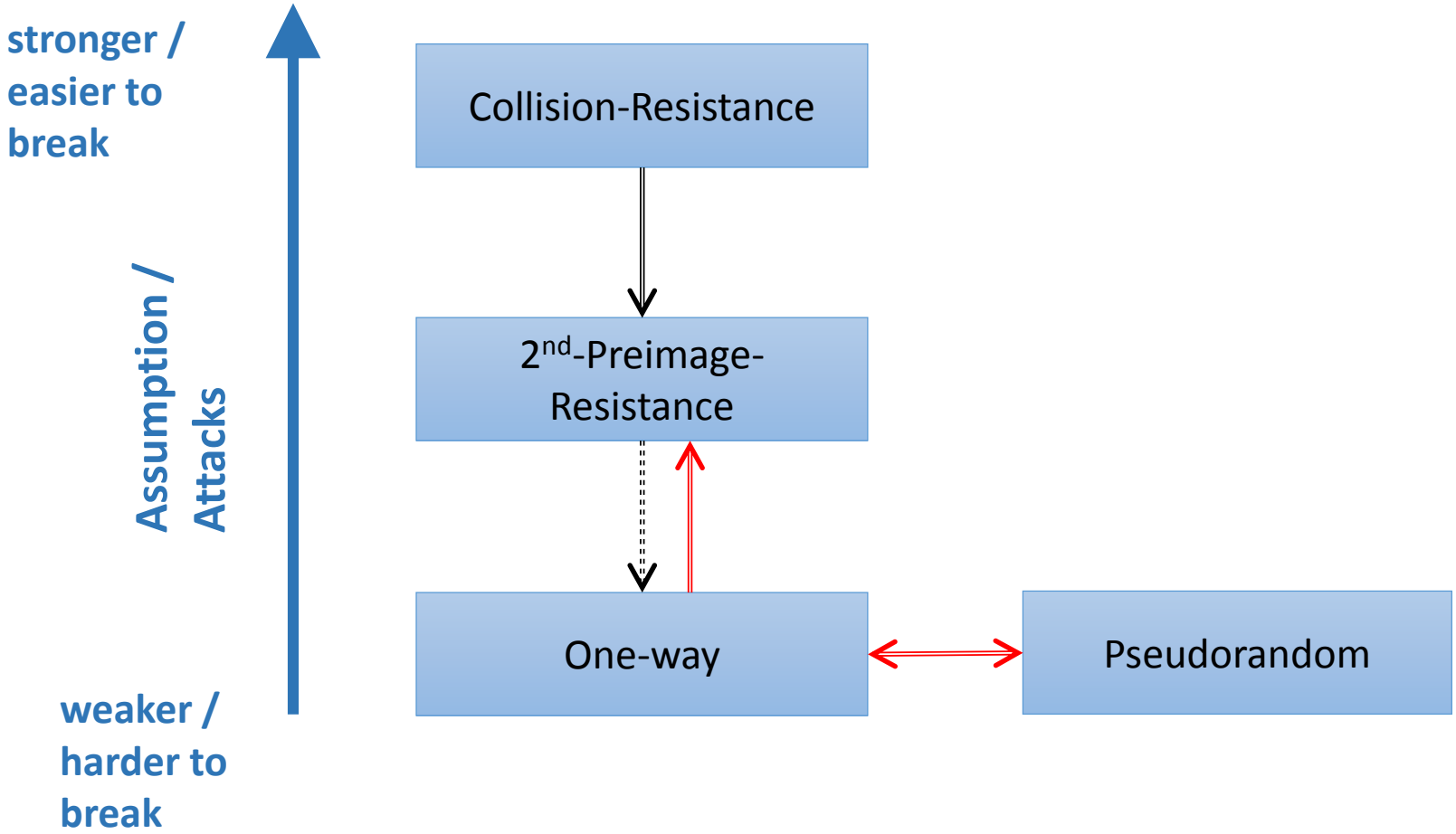
(Disclaimer: Currently only proof for $2^m \gg 2^n$)

Generic Security

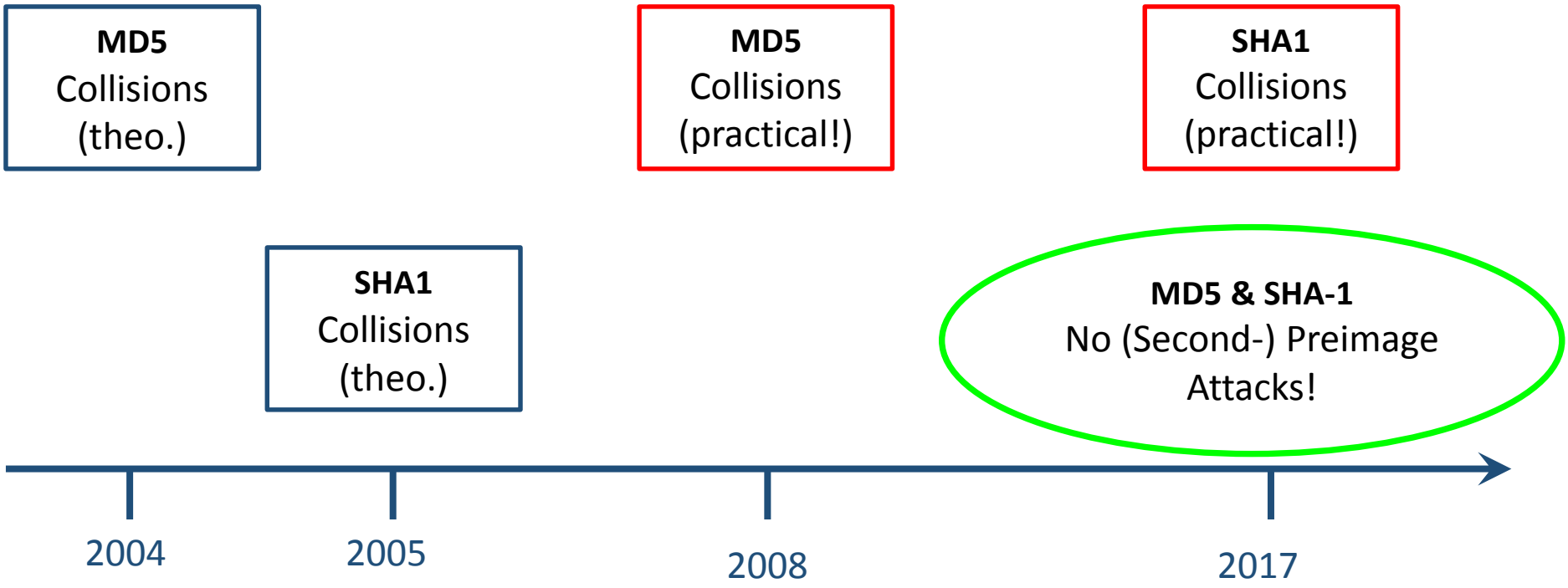
	OW	SPR	CR	UD*	PRF*
Classical	$\Theta(2^n)$	$\Theta(2^n)$	$\Theta(2^{n/2})$	$\Theta(2^n)$	$\Theta(2^n)$
Quantum	$\Theta(2^{n/2})$	$\Theta(2^{n/2})$	$\Theta(2^{n/3})$	$\Theta(2^{n/2})$	$\Theta(2^{n/2})$

* conjectured, no proof

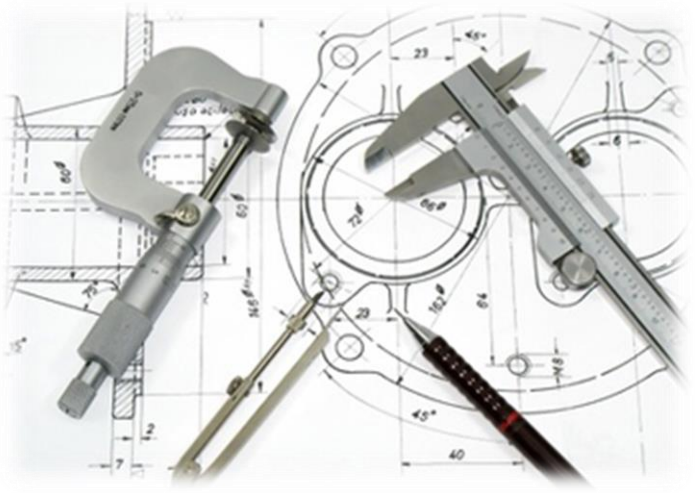
Hash-function properties



Attacks on Hash Functions

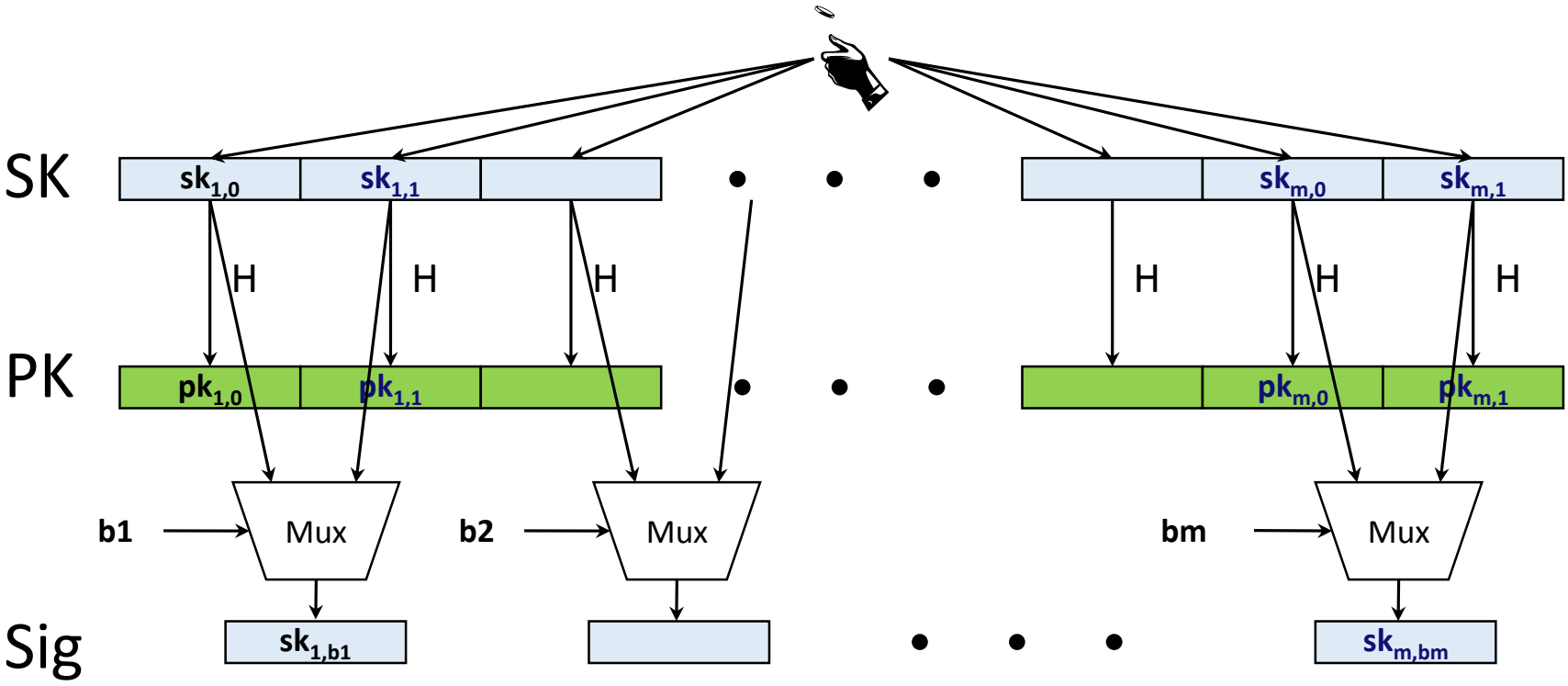


Basic Construction

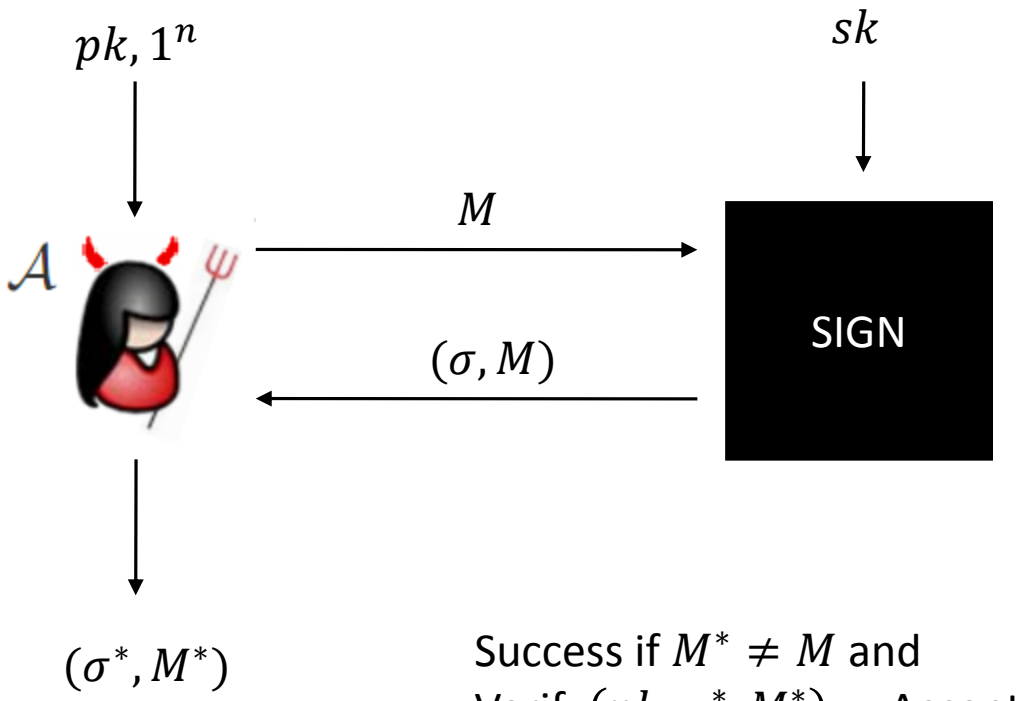


Lamport-Diffie OTS [Lam79]

Message M = b1,…,bm, OWF H * = n bit



EU-CMA for OTS



Success if $M^* \neq M$ and
Verify$(pk, \sigma^*, M^*) =$ Accept

Security

Theorem:

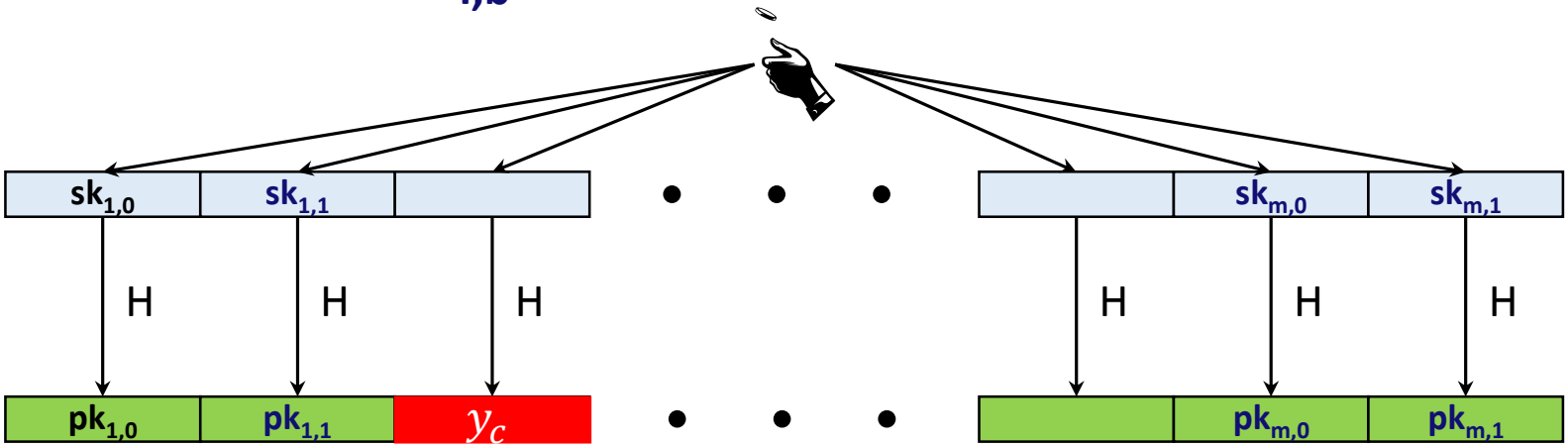
If H is one-way then LD-OTS is one-time eu-cma-secure.

Reduction

Input: y_c, k

Set $H \leftarrow h_k$

Replace random $\mathbf{pk_{i,b}}$



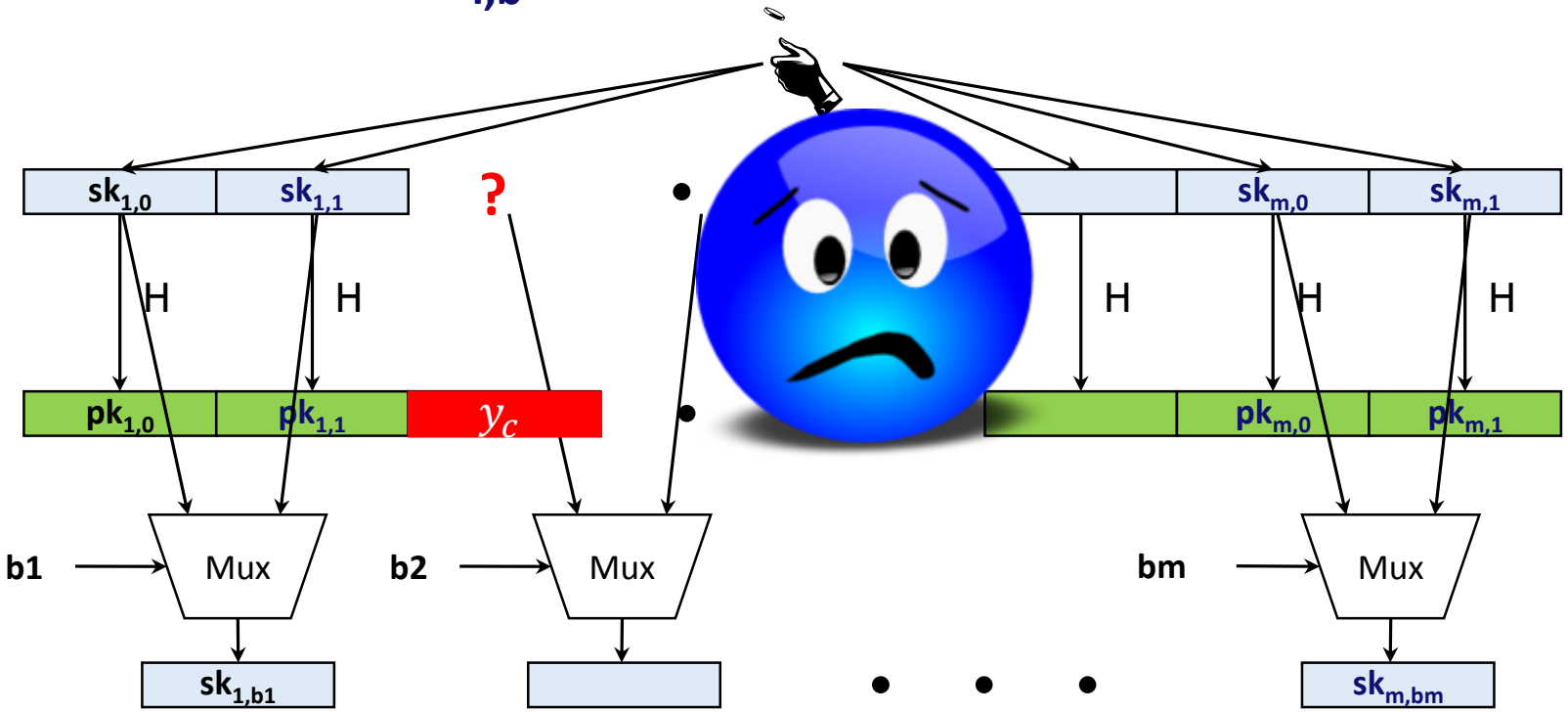
Reduction

Input: y_c, k

Set $H \leftarrow h_k$

Replace random $\mathbf{pk_{i,b}}$

Adv. Message: M = b1,…,bm
If bi = b return fail
else return Sign(M)



Reduction

Input: y_c, k

Set $H \leftarrow h_k$

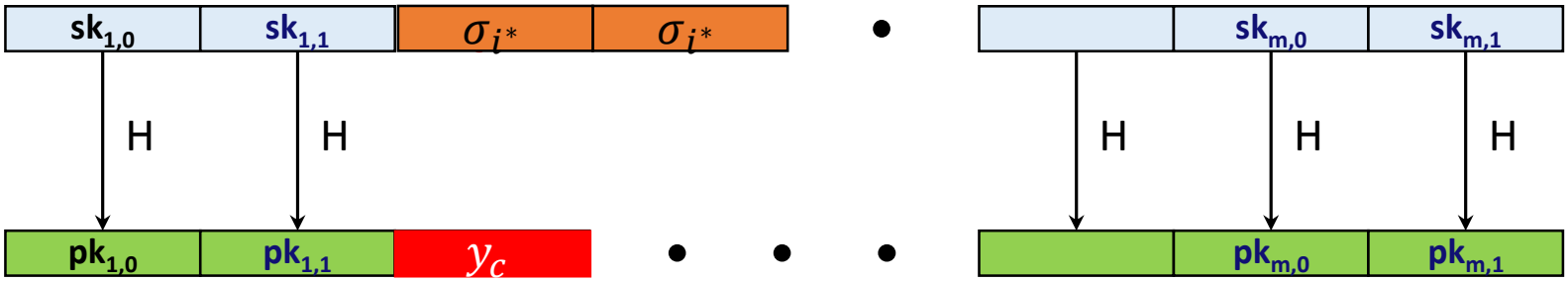
Choose random $\mathbf{pk_{i,b}}$

Forgery: M* = b1*,…,bm*,

$$\sigma = \sigma_1, \ldots, \sigma_m$$

If bi ≠ b return fail

Else return σ_{i^*}



Reduction - Analysis

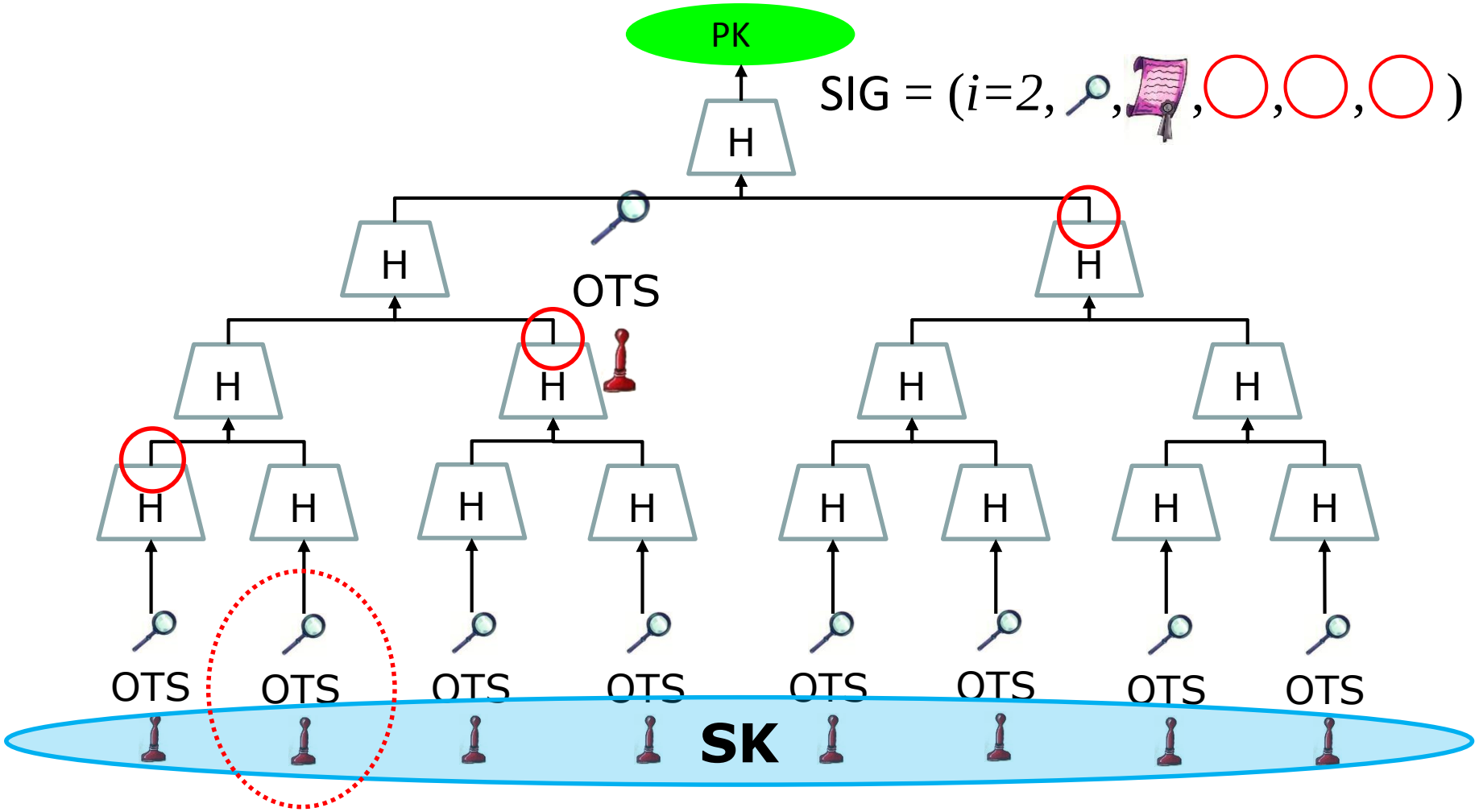
Abort in two cases:

1. $b_i = b$
   probability ½ : b is a random bit
2. $b_i^* \neq b$

   probability 1 - 1/m: At least one bit has to flip as $M^* \neq M$

Reduction succeeds with A's success probability times 1/2m.

Merkle's Hash-based Signatures



Security

Theorem:

MSS is eu-cma-secure if OTS is a one-time eu-cma secure signature scheme and H is a random element from a family of collision resistant hash functions.

Reduction

Input: k, pk_{OTS}

1. Choose random $0 \leq i < 2^h$
2. Generate key pair using pk_{OTS} as ith OTS public key and $H \leftarrow h_k$
3. Answer all signature queries using sk or sign oracle (for index i)
4. Extract OTS-forgery or collision from forgery

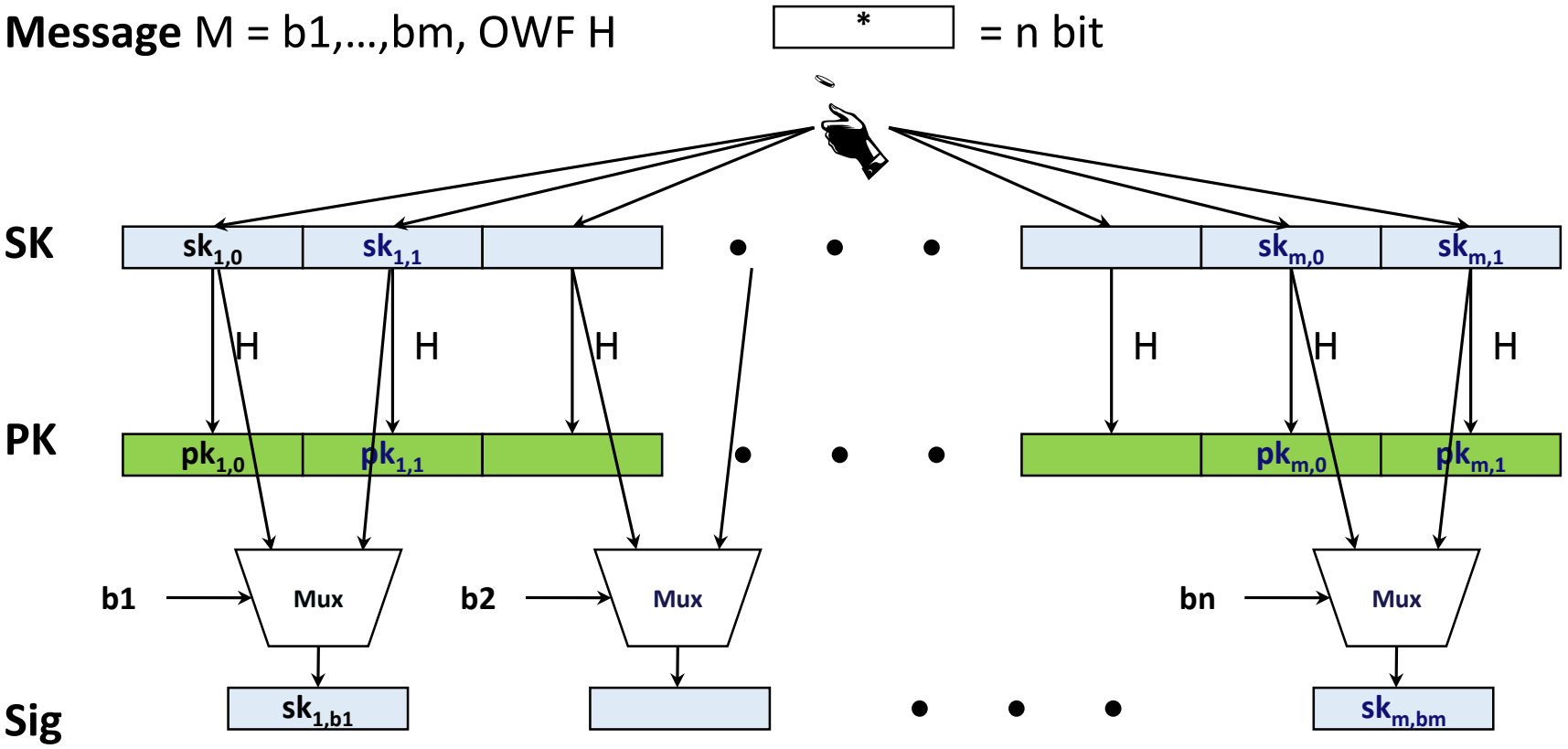
Reduction (Step 4, Extraction)

Forgery: ($i^*, \sigma^*_{OTS}, pk^*_{OTS}$, AUTH)

1. If pk^*_{OTS} equals OTS pk we used for i^* OTS, we got an OTS forgery.
   - Can only be used if $i^* = i$.
2. Else adversary used different OTS pk.
   - Hence, different leaves.
   - Still same root!
   - Pigeon-hole principle: Collision on path to root.

Winternitz-OTS

Recap LD-OTS [Lam79]



LD-OTS in MSS

SIG = (i=2,  ,  , ◯, ◯, ◯)

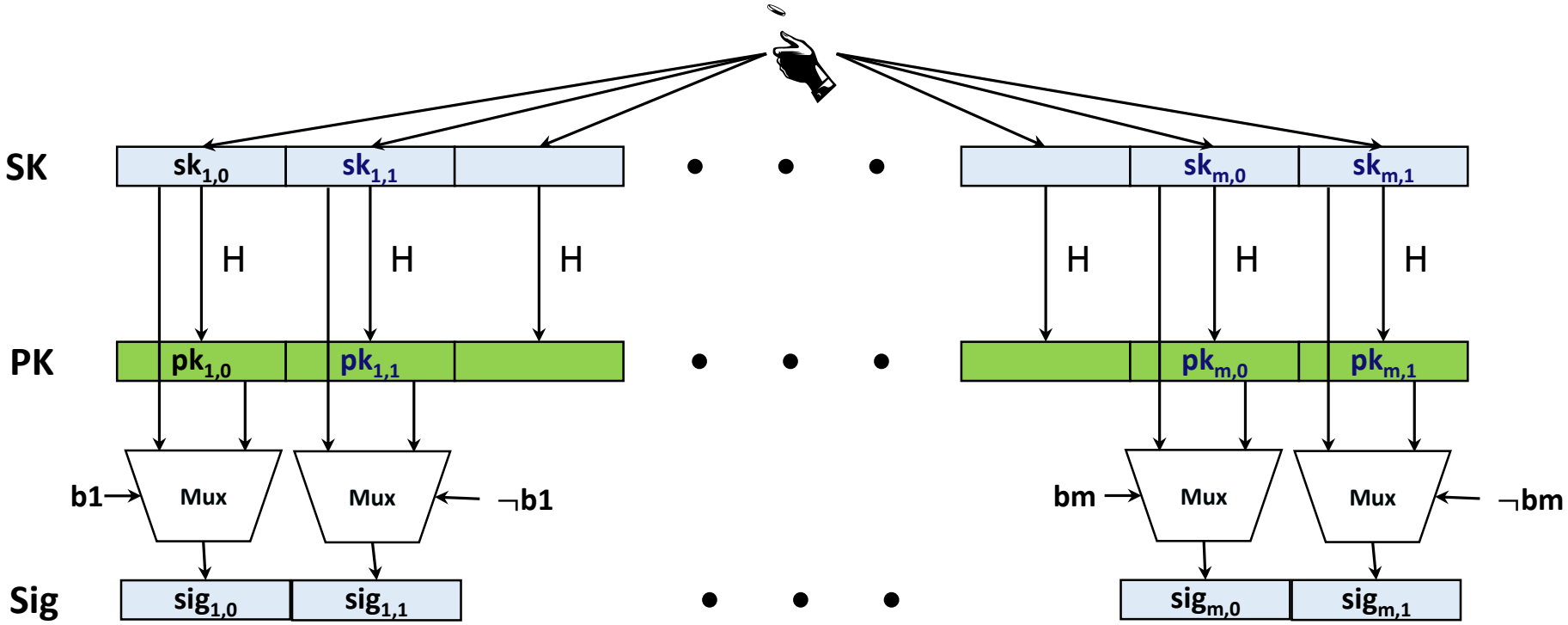
Verification:

1. Verify 
2. Verify authenticity of 

We can do better!

Trivial Optimization

Message M = b1,…,bm, OWF H * = n bit



Optimized LD-OTS in MSS

SIG = (i=2,  , ○, ○, ○)

Verification:

1. Compute  from 
2. Verify authenticity of 

Steps 1 + 2 together verify 

Let‘s sort this

Message $M = b_1,\ldots,b_m$, OWF H

SK: $sk_1,\ldots,sk_m,sk_{m+1},\ldots,sk_{2m}$

PK: $H(sk_1),\ldots,H(sk_m),H(sk_{m+1}),\ldots,H(sk_{2m})$

Encode M: $M' = M||\neg M = b_1,\ldots,b_m,\neg b_1,\ldots,\neg b_m$

(instead of $b_1, \neg b_1,\ldots,b_m,\neg b_m$)

Sig: $sig_i =$

$$\begin{cases} sk_i & \text{, if } b_i = 1 \\ H(sk_i) & \text{, otherwise} \end{cases}$$

Checksum with bad performance!

Optimized LD-OTS

Message M = $b_1,\ldots,b_m$, OWF H

SK: $sk_1,\ldots,sk_m,sk_{m+1},\ldots,sk_{m+1+\log m}$

PK: $H(sk_1),\ldots,H(sk_m),H(sk_{m+1}),\ldots,H(sk_{m+1+\log m})$

Encode M: M‘ = $b_1,\ldots,b_m,\neg \sum_1^m b_i$

Sig: sig_i =

$$\begin{cases} sk_i & \text{, if } b_i = 1 \\ H(sk_i) & \text{, otherwise} \end{cases}$$

IF one b_i is flipped from 1 to 0, another b_j will flip from 0 to 1

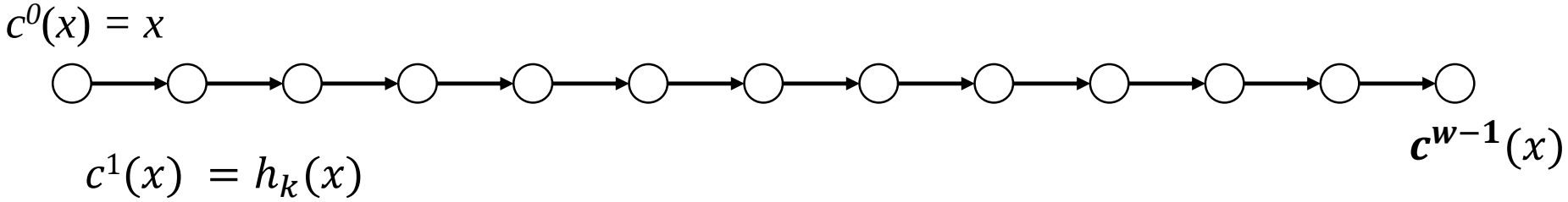
Function chains

Function family: $H_n := \{h_k : \{0,1\}^n \rightarrow \{0,1\}^n\}$

$h_k \xleftarrow{\$} H_n$

Parameter w

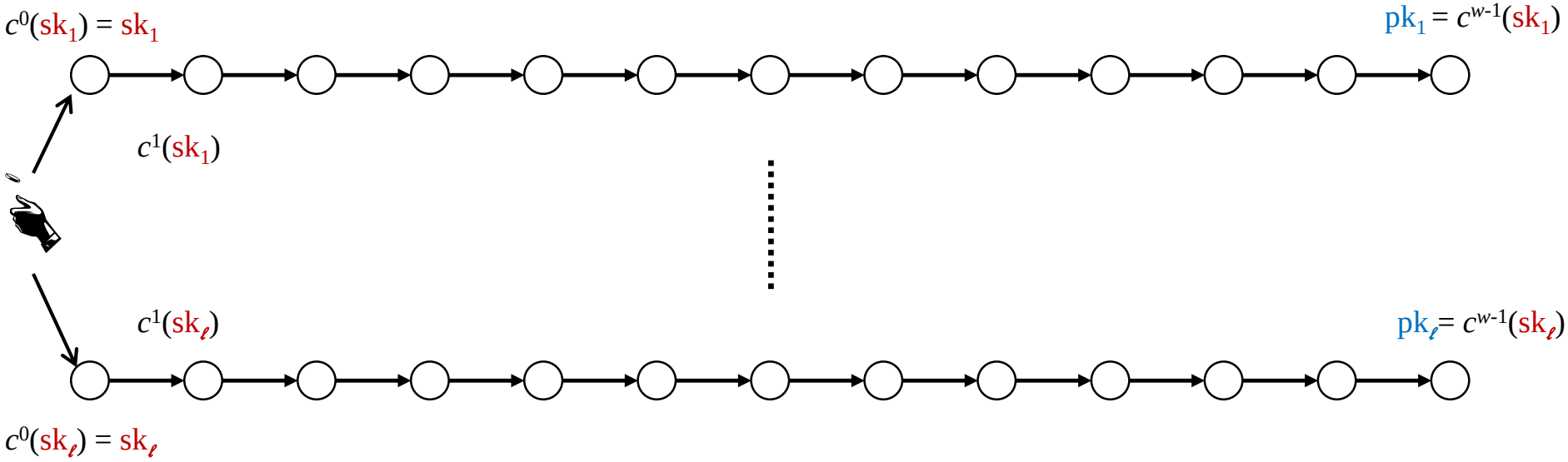
Chain: $c^i(x) = h_k(c^{i-1}(x)) = \underbrace{h_k \circ h_k \circ \ldots \circ h_k(x)}_{i-times}$



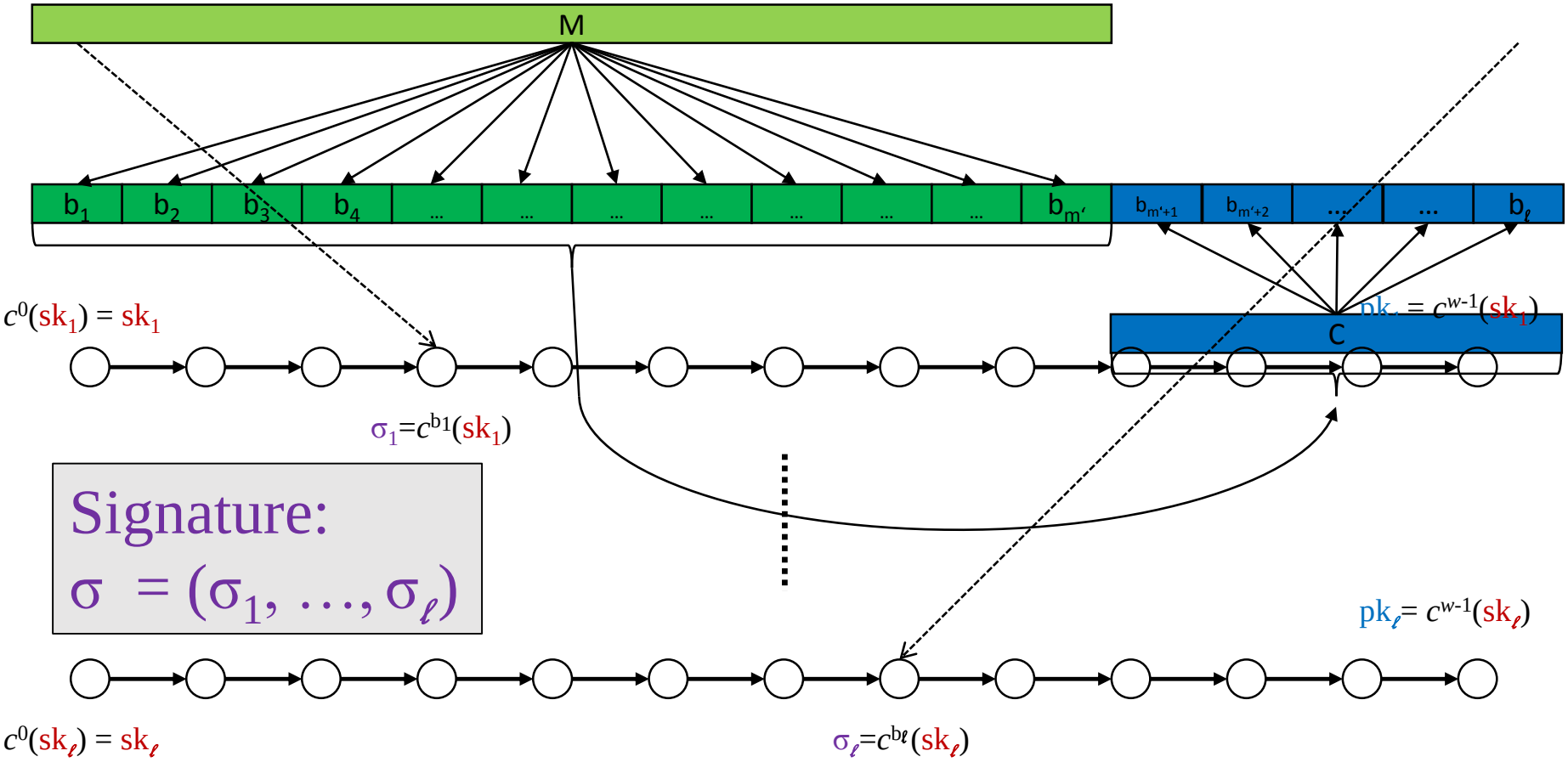
WOTS

Winternitz parameter $\boldsymbol{w}$, security parameter $\boldsymbol{n}$, message length $\boldsymbol{m}$, function family H_n

Key Generation: Compute l, sample h_k

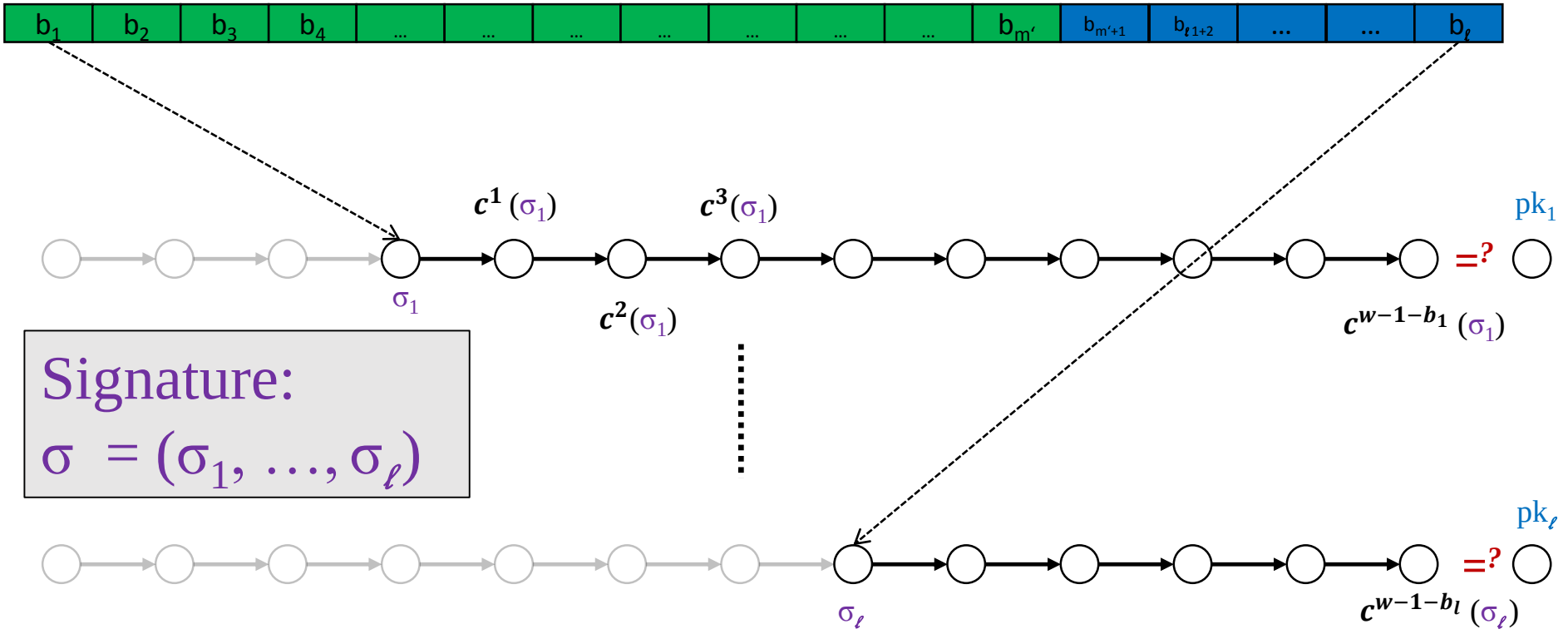


WOTS Signature generation



WOTS Signature Verification

Verifier knows: M, w



WOTS Function Chains

For $x \in \{0,1\}^n$ define $c^0(x) = x$ and

- WOTS: $c^i(x) = h_k(c^{i-1}(x))$
- WOTS$^{\$}$: $c^i(x) = h_{c^{i-1}(x)}(r)$
- WOTS^{+}: $c^i(x) = h_k(c^{i-1}(x) \oplus r_i)$

WOTS Security

Theorem (informally):

W-OTS *is strongly unforgeable under chosen message attacks if* H_n *is a collision resistant family of undetectable one-way functions.*

W-OTS$^{\$}$ *is existentially unforgeable under chosen message attacks if* H_n *is a pseudorandom function family.*

W-OTS^{+} *is strongly unforgeable under chosen message attacks if* H_n *is a* 2^{nd}*-preimage resistant family of undetectable one-way functions.*

XMSS

XMSS

Tree: Uses bitmasks

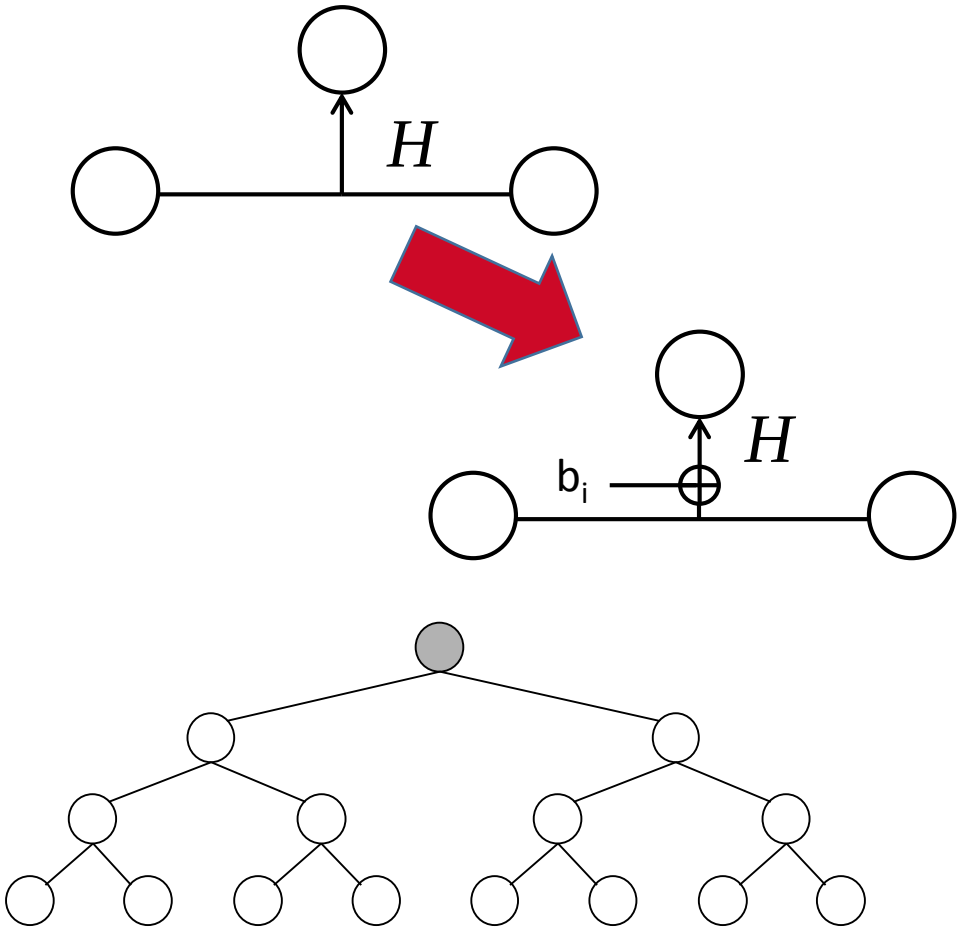
Leafs: Use binary tree with bitmasks

OTS: WOTS^{+}

Mesage digest: Randomized hashing

Collision-resilient

-> signature size halved

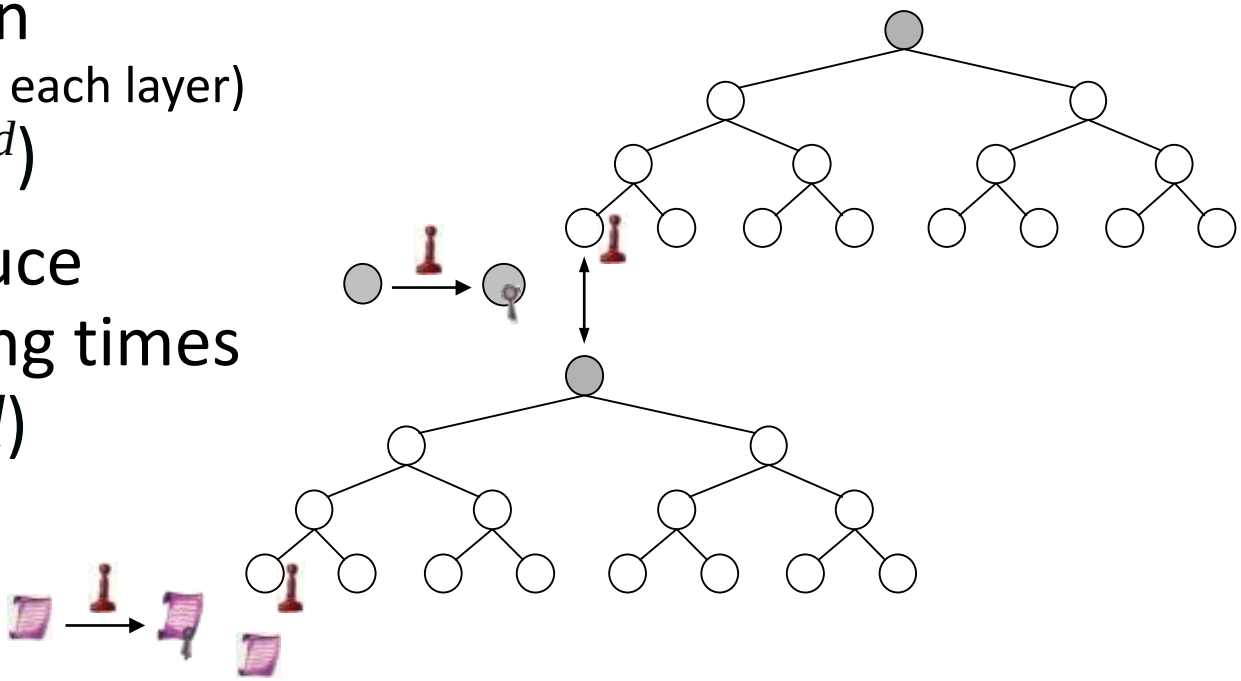


Multi-Tree XMSS

Uses multiple layers of trees

-> Key generation
(= Building first tree on each layer)
$\Theta(2^h) \rightarrow \Theta(d*2^{h/d})$

-> Allows to reduce worst-case signing times
$\Theta(\mathrm{h}/2) \rightarrow \Theta(h/2d)$



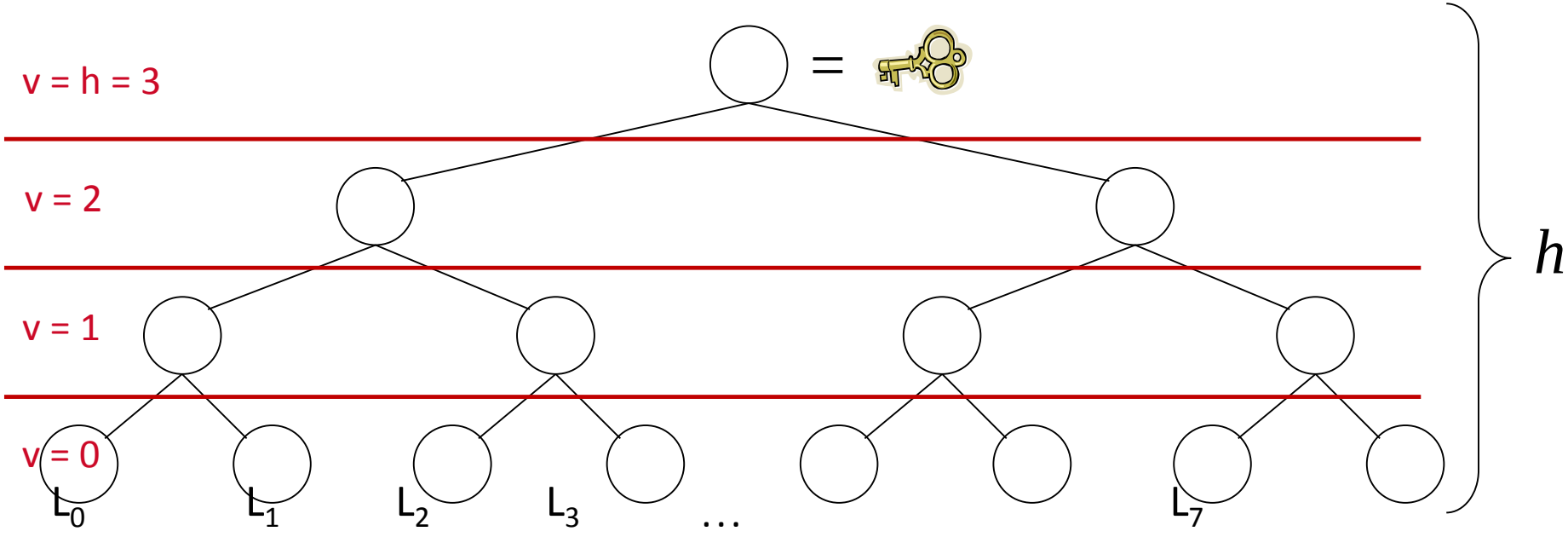
Authentication path computation

TreeHash

(Mer89)

TreeHash

- TreeHash(v,i): Computes node on level v with leftmost descendant L_i
- Public Key Generation: Run TreeHash(h,0)

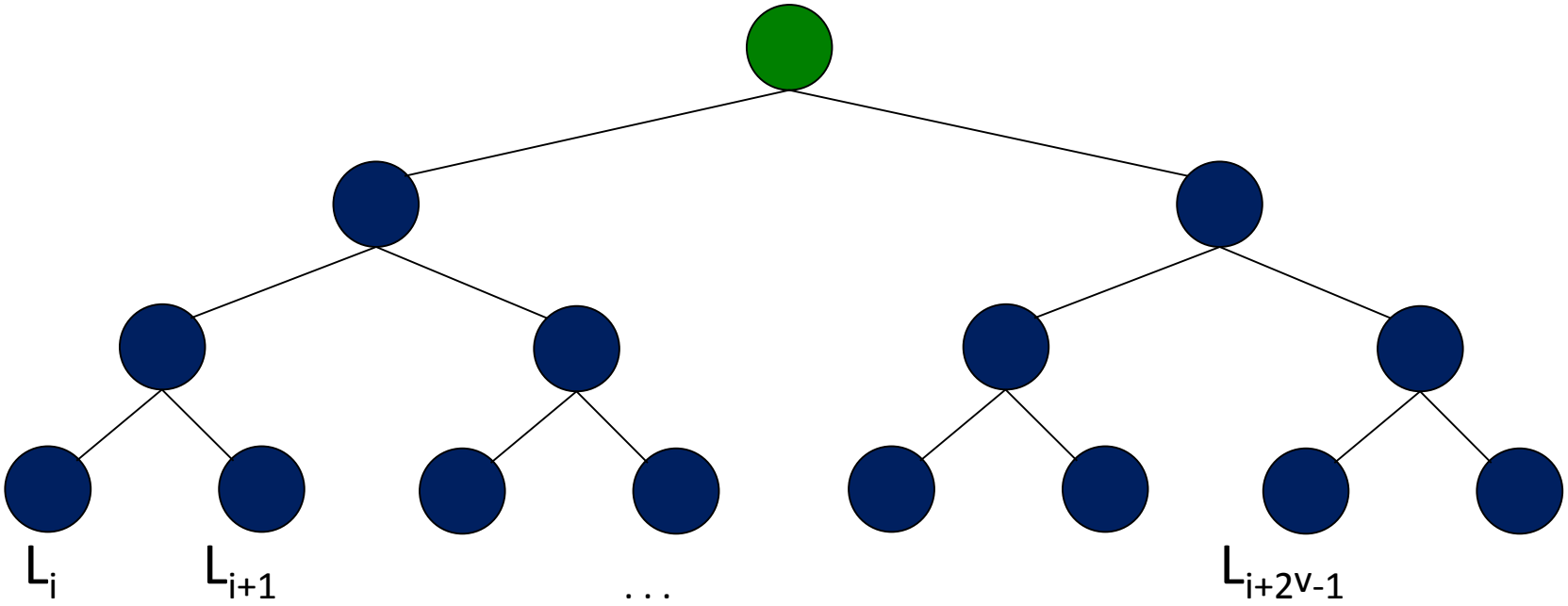


TreeHash

```
TreeHash(v,i)
1: Init Stack, N1, N2
2: For j = i to i+2^v-1 do
3:         N1 = LeafCalc(j)
4:         While N1.level() == Stack.top().level() do
5:                 N2 = Stack.pop()
6:                 N1 = ComputeParent( N2, N1 )
7:         Stack.push(N1)
8: Return Stack.pop()
```

TreeHash

TreeHash(v,i)



Efficiency?

Key generation: Every node has to be computed once.

cost = 2^h leaves + 2^h-1 nodes

=> optimal

Signature: One node on each level 0 <= v < h.

cost 2^h-1 leaves + 2^h-1-h nodes.

Many nodes are computed many times!

(e.g. those on level v=h-1 are computed 2^{h-1} times)

-> Not optimal if state allowed

The BDS Algorithm

[BDS08]

Motivation
(for all Tree Traversal Algorithms)

No Storage:
Signature: Compute one node on each level 0 <= v < h.
Costs: 2^h-1 leaf + 2^h-1-h node computations.

Example: XMSS with SHA2-256 and h = 20 -> approx. 15min

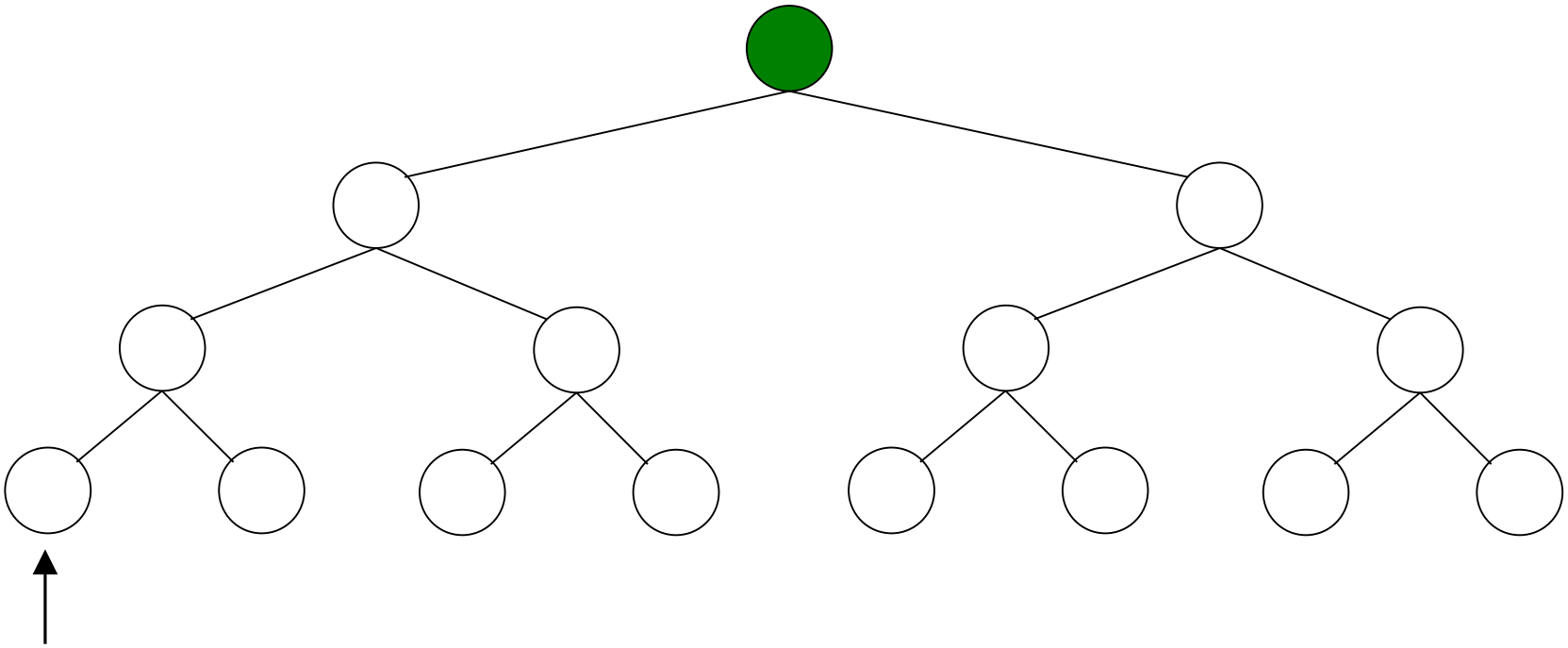
Store whole tree: 2^hn bits.

Example: h=20, n=256; storage: 2^{28}bits = 32MB

Idea: Look for time-memory trade-off!

Use a State

Authentication Paths



Observation 1

Same node in authentication path is recomputed many times!

Node on level v is recomputed for 2^v successive paths.

Idea: Keep authentication path in state.

-> Only have to update "new" nodes.

Result

Storage: h nodes

Time: ~ h leaf + h node computations (average)

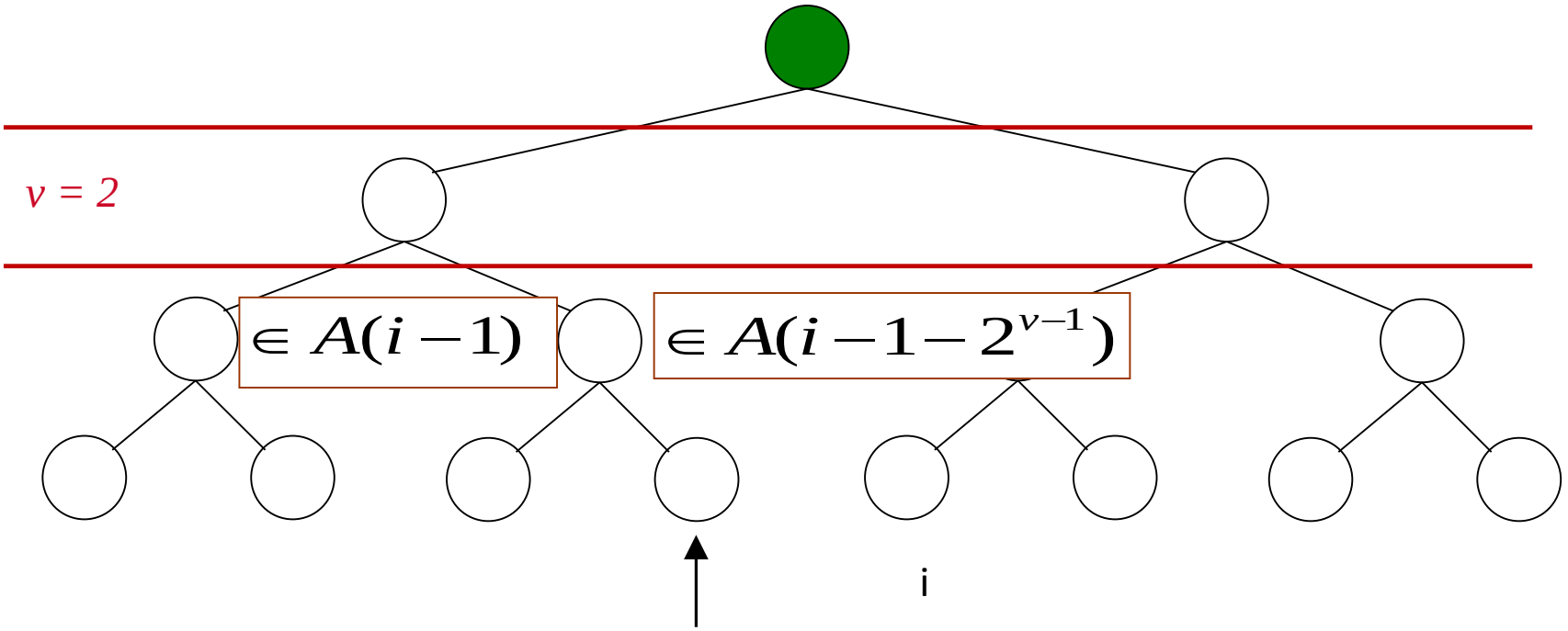
But: Worst case still 2^h-1 leaf + 2^h-1-h node computations!

-> Keep in mind. To be solved.

Observation 2

When new left node in authentication path is needed, its children have been part of previous authentication paths.

Computing Left Nodes



Result

Storing $\left\lceil \frac{h}{2} \right\rceil$ nodes

all left nodes can be computed with one node computation / node

Observation 3

Right child nodes on high levels are most costly.

Computing node on level v requires
2^v leaf and 2^v-1 node computations.

Idea: Store right nodes on top k levels during key generation.

Result
Storage: 2^k-2 n bit nodes
Time: ~ h-k leaf + h-k node computations (average)

Still: Worst case 2^{h-k}-1 leaf + 2^{h-k}-1-(h-k) node computations!

Distribute Computation

Intuition

Observation:

- For every second signature only one leaf computation
- Average runtime: ~ h-k leaf + h-k node computations

Idea: Distribute computation to achieve average runtime in worst case.

Focus on distributing computation of leaves

TreeHash with Updates

TreeHash.init(v,i)

1: Init Stack, N1, N2, j=i, j_max = i+2^v-1

2: Exit

TreeHash.update()

1: If j <= j_max

2: N1 = LeafCalc(j)

3: While N1.level() == Stack.top().level() do

5: N2 = Stack.pop()

6: N1 = ComputeParent(N2, N1)

7: Stack.push(N1)

8: Set j = j+1

9: Exit

One leaf per update

Distribute Computation

Concept

- Run one TreeHash instance per level 0 <= v < h-k
- Start computation of next right node on level v when current node becomes part of authentication path.
- Use scheduling strategy to guarantee that nodes are finished in time.
- Distribute (h-k)/2 updates per signature among all running TreeHash instances

Distribute Computation

Worst Case Runtime

Before:
2^{h-k}-1 leaf and 2^{h-k}-1-(h-k) node computations.

With distributed computation:
(h-k)/2 + 1 leaf and 3(h-k-1)/2 + 1 node computations.

Add. Storage

Single stack of size h-k nodes for all TreeHash instances.
+ One node per TreeHash instance.
= 2(h-k) nodes

BDS Performance

Storage:

$$3h + \left\lfloor \frac{h}{2} \right\rfloor - 3k - 2 + 2^k \quad n \text{ bit nodes}$$

Runtime:

$(h-k)/2+1$ leaf and

$3(h-k-1)/2+1$ node computations.

XMSS in practice

XMSS Internet-Draft

(draft-irtf-cfrg-xmss-hash-based-signatures)

- Protecting against multi-target attacks / tight security
  - n-bit hash => n bit security
- Small public key (2n bit)
  - At the cost of ROM for proving PK compression secure
- Function families based on SHA2

- Equal to XMSS-T [HRS16] up-to message digest

XMSS / XMSS-T Implementation

C Implementation, using OpenSSL [HRS16]

	Sign (ms)	Signature (kB)	Public Key (kB)	Secret Key (kB)	Bit Security classical/ quantum	Comment
XMSS	3.24	2.8	1.3	2.2	236 / 118	h = 20, d = 1,
XMSS-T	9.48	2.8	**0.064**	2.2	**256 / 128**	h = 20, d = 1
XMSS	3.59	8.3	1.3	14.6	196 / 98	h = 60, d = 3
XMSS-T	10.54	8.3	**0.064**	14.6	**256 / 128**	h = 60, d = 3

Intel(R) Core(TM) i7 CPU @ 3.50GHz
XMSS-T uses message digest from Internet-Draft
All using SHA2-256, w = 16 and k = 2

Open research topics

1. Message compression which
   - is collision-resilient,
   - provdies tight provable security,
   - especially resists multi-target attacks (=> no eTCR)
   - => Has applications outside hash-based crypto!
2. Quantum query complexity for further properties
   - E.g. PRF, UD, aSec, ...
3. Quantum security of existing hash function constructions.
   - E.g. can classical attacks be improved (e.g. differential cryptanalysis)
   - Formal proofs (see recent works on collapsing hashes)

SPHINCS

About the statefulness

- Works great for some settings

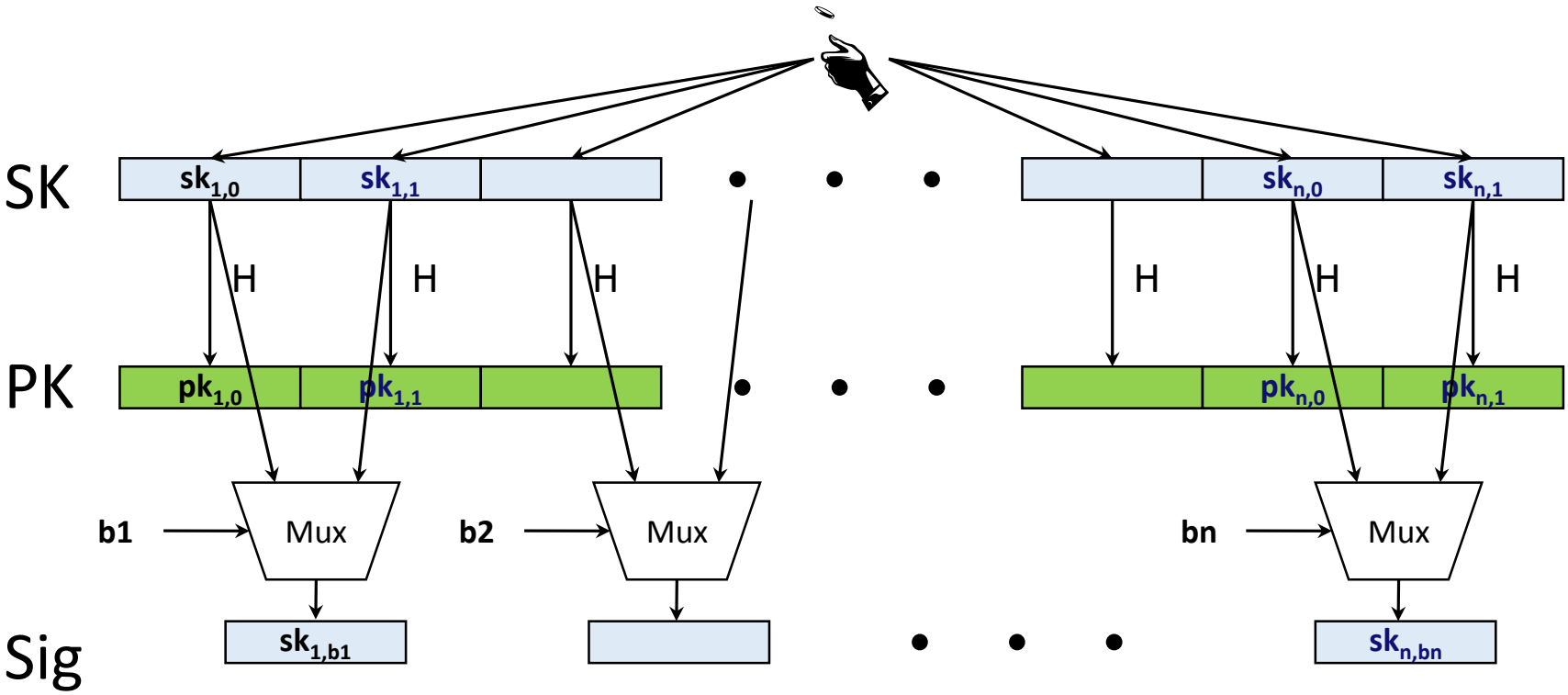
- However....
  - ... back-up
  - ... multi-threading
  - ... load-balancing


ELIMINATE
THE STATE

Few-Time Signature Schemes

Recap LD-OTS

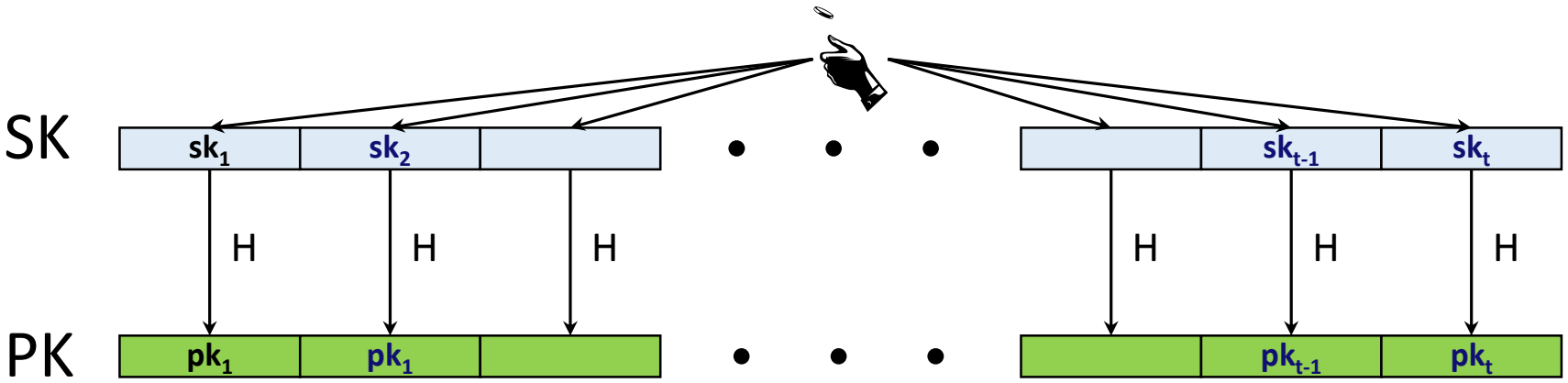
Message M = b1,…,bn, OWF H * = n bit



HORS [RRO2]

Message M, OWF H, CRHF H’ * = n bit

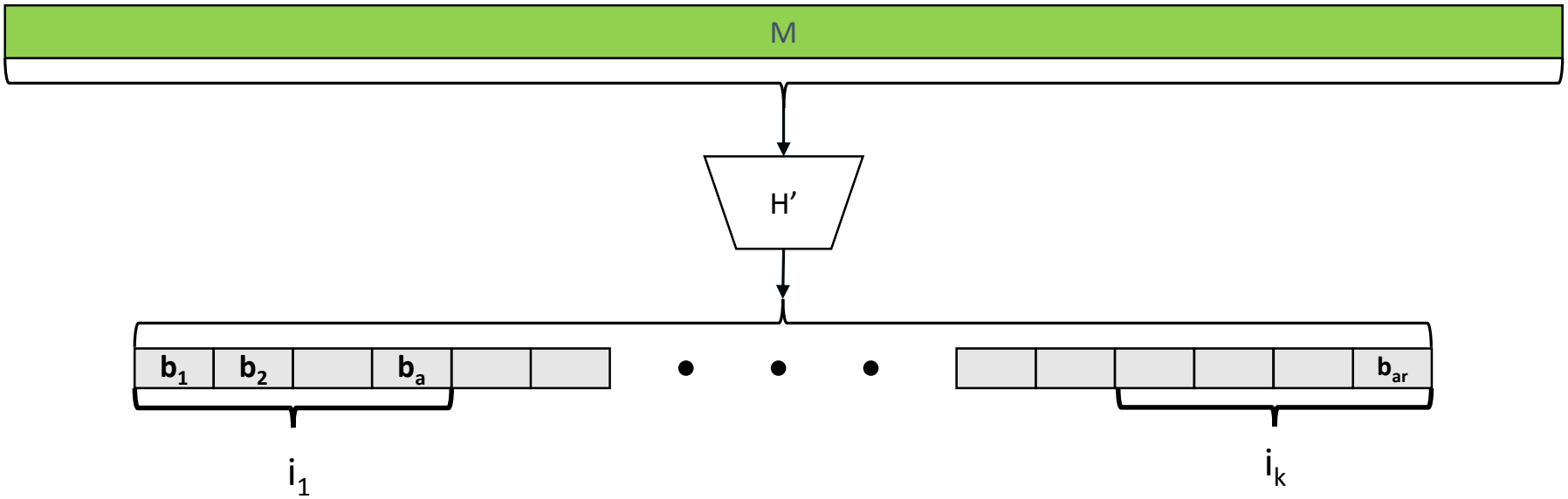
Parameters $t=2^a$,k, with m = ka (typical a=16, k=32)



HORS mapping function

Message M, OWF H, CRHF H’ * = n bit

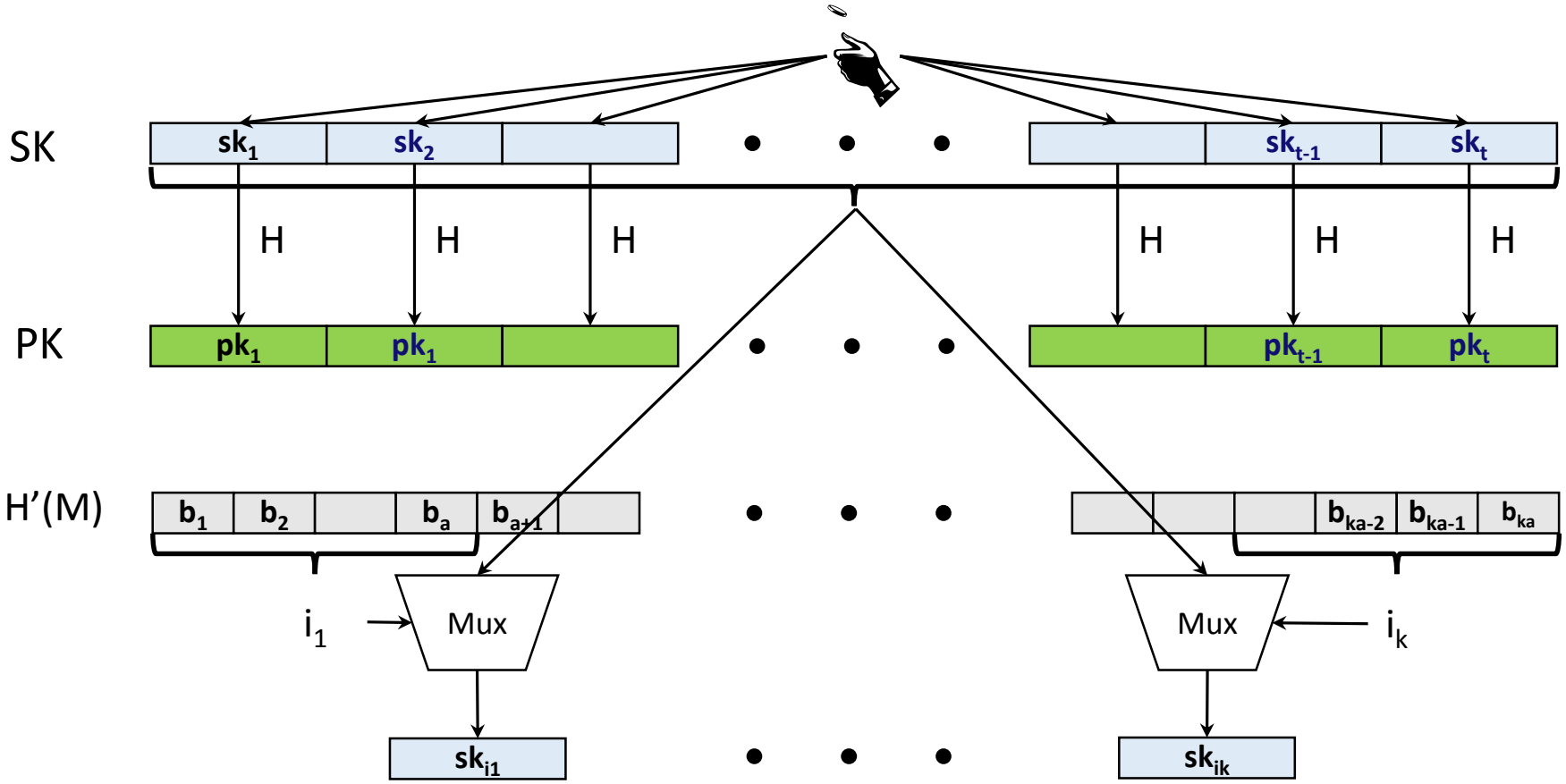
Parameters $t=2^a$,k, with m = ka (typical a=16, k=32)



HORS

Message M, OWF H, CRHF H’ [*] = n bit

Parameters $t=2^a$,k, with m = ka (typical a=16, k=32)



HORS Security

- M mapped to k element index set $M^i \in \{1, \ldots, t\}^k$
- Each signature publishes k out of t secrets
- Either break one-wayness or…

- r-Subset-Resilience: After seeing index sets M_j^i for r messages $msg_j, 1 \leq j \leq r$, hard to find $msg_{r+1} \neq msg_j$ such that $M_{r+1}^i \in \bigcup_{1 \leq j \leq r} M_j^i$.

- Best generic attack: $\text{Succ}_{\text{r-SSR}}(A, q) = q \left(\frac{rk}{t}\right)^k$

$\longrightarrow$ Security shrinks with each signature!

HORST

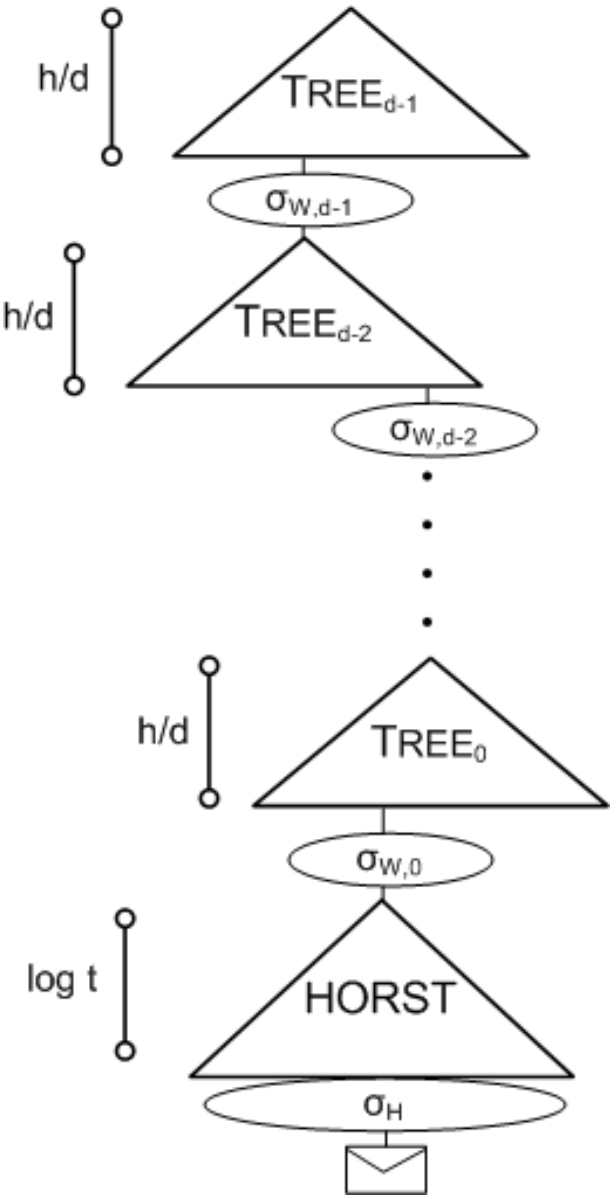
Using HORS with MSS requires adding PK (tn) to MSS signature.

HORST: Merkle Tree on top of HORS-PK

- New PK = Root
- Publish Authentication Paths for HORS signature values
- PK can be computed from Sig
- With optimizations: tn $\longrightarrow$ (k(log t − x + 1) + 2^x)n
  - E.g. SPHINCS-256: 2 MB $\longrightarrow$ 16 KB
- Use randomized message hash

SPHINCS

- Stateless Scheme
- XMSS^{MT} + HORST
  + (pseudo-)random index
- Collision-resilient
- Deterministic signing
- SPHINCS-256:
  - 128-bit post-quantum secure
  - Hundrest of signatures / sec
  - 41 kb signature
  - 1 kb keys

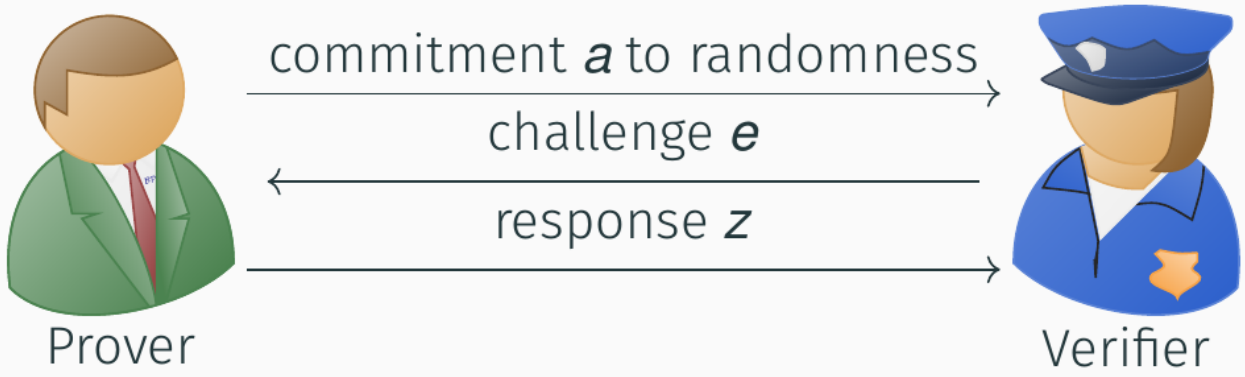


Signatures via Non-Interactive Proofs: The Case of Fish & Picnic

Thanks to the Fish/Picnic team for slides

Interactive Proofs

Three move protocol:



- Important that e unpredictable before sending a
- aka (Interactive) Honest-Verifier Zero-Knowledge Proofs

Non-interactive variant via Fiat-Shamir [FS86] transform

ZKBoo

Efficient Σ-protocols for arithmetic circuits

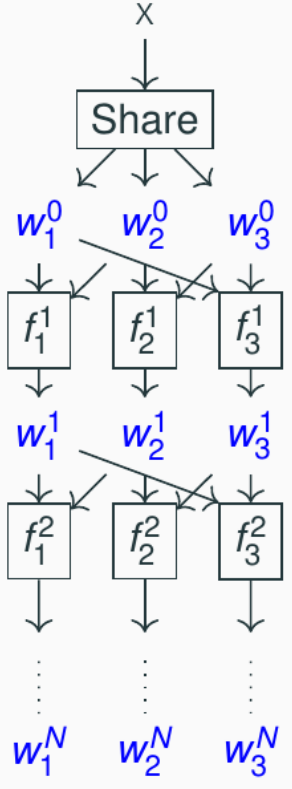
- generalization, simplification, + implementation of "MPC-in-the-head" [IKOS07]

Idea

1. (2,3)-decompose circuit into three shares
2. Revealing 2 parts reveals no information
3. Evaluate decomposed circuit per share
4. Commit to each evaluation
5. Challenger requests to open 2 of 3
6. Verifies consistency

Efficiency

- Heavily depends on #multiplications



High-Level Approach

- Use LowMC v2 to build dedicated hash function with low AND-gate-depth
- Use ZKBoo to proof knowledge of a preimage
- Use Fiat-Shamir to turn ZKP into Signature in ROM (Fish), or
- Use Unruh‘s transform to turn ZKP into Signature in QROM (Picnic)

Performance

Scheme	Gen	Sign	Verify	$\|sk\|$	$\|pk\|$	$\|\sigma\|$	M
Fish-10-38	0.01	29.73	17.46	32	32/64	116*K*	ROM
Picnic-10-38	0.01	31.31	16.30	32	32/64	191*K*	QROM
MQ 5pass	1.0	7.2	5.0	32	74	40*K*	ROM
SPHINCS-256	0.8	1.0	0.6	1*K*	1*K*	40*K*	SM
BLISS-I	44	0.1	0.1	2*K*	7*K*	5.6*K*	ROM
Ring-TESLA	17*K*	0.1	0.1	12*K*	8*K*	1.5*K*	ROM
TESLA-768	49*K*	0.6	0.4	3.1*M*	4*M*	2.3*K*	(Q)ROM
FS-Véron	n/a	n/a	n/a	32	160	$\geq$ 126*K*	ROM
SIDHp751	16	7*K*	5*K*	48	768	138*K*	QROM

Table 2: Timings (ms) and key/signature sizes (bytes)

Open research topics II

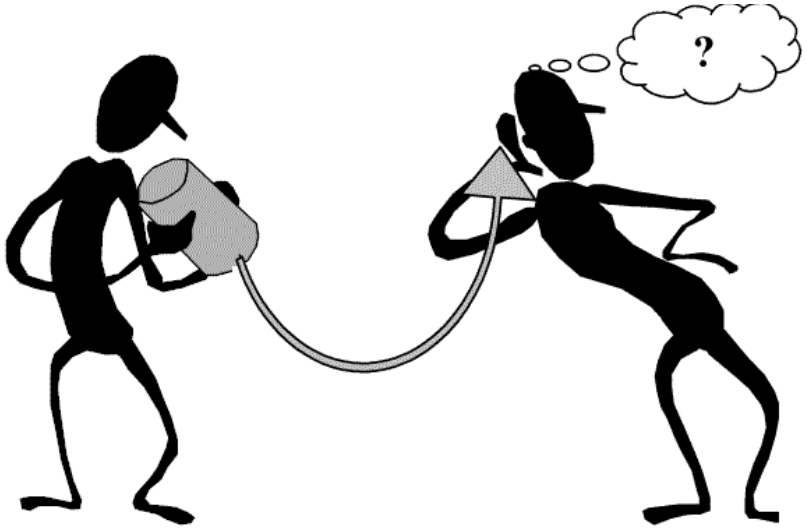
SPHINCS:

- More efficient few-time signatures
- Dedicated fast short, constant size input hash functions.

Fish / Picnic

- More efficient (size!!!) QROM transform
- Dedicated, more efficient proof for knowledge of preimage?
- Hash functions with lower AND-gate depth.

Thank you!
Questions?



For references & further literature see
https://huelsing.wordpress.com/hash-based-signature-schemes/literature/