Hash-based Signatures

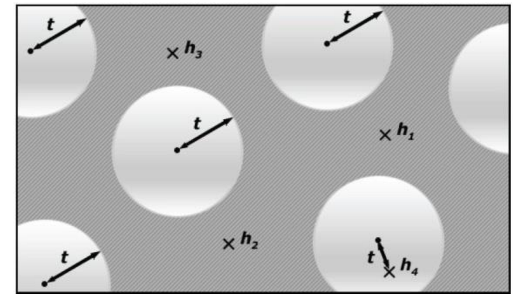
Andreas Hülsing
Eindhoven University of Technology

Executive School on Post-Quantum Cryptography
July 2019, TU Eindhoven

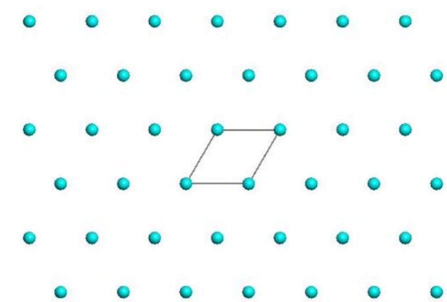
Post-Quantum Signatures

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

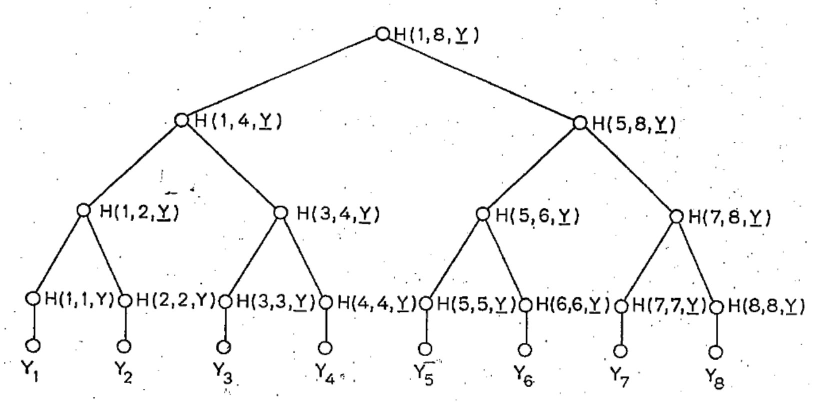
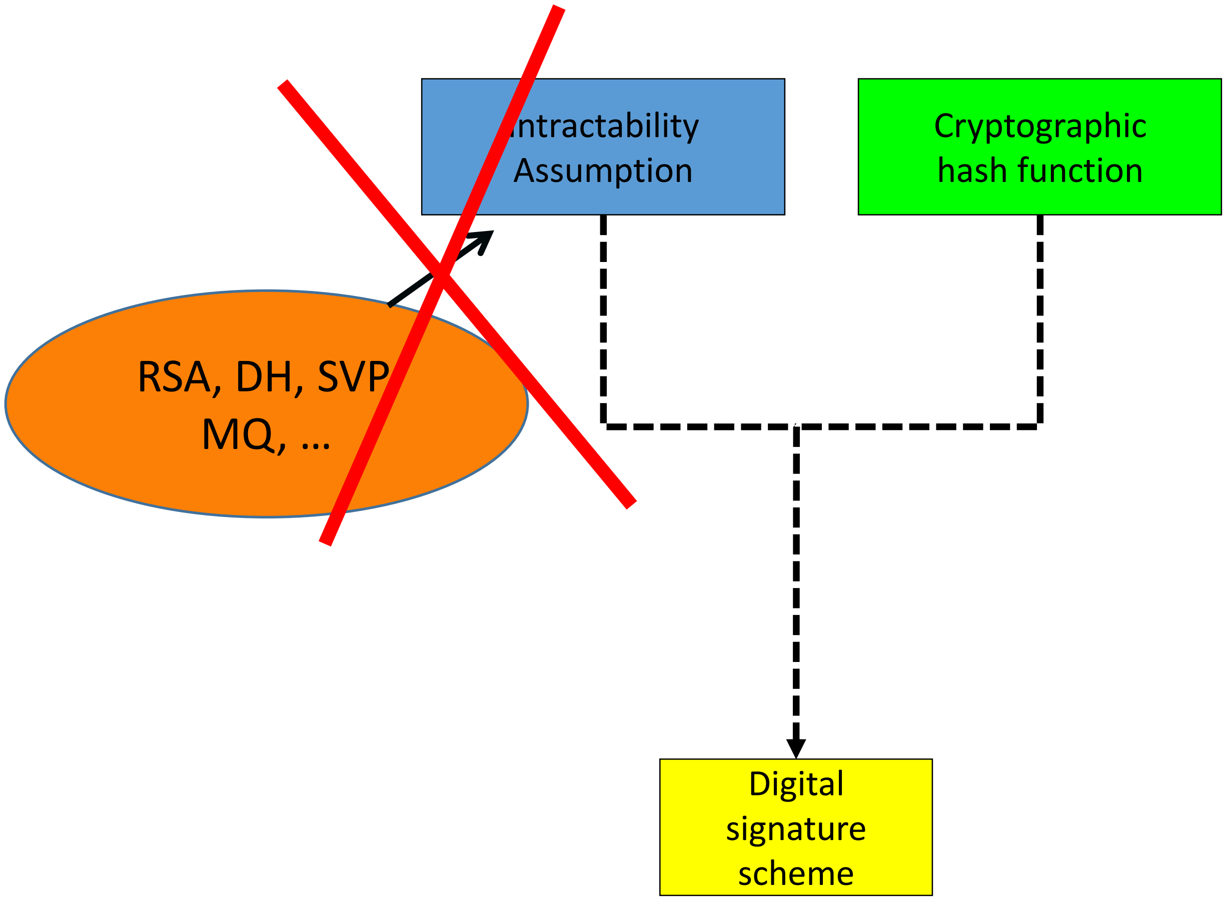


FIG 1

AN AUTHENTICATION TREE WITH N = 8.

PAGE 41B

RSA – DSA – EC-DSA...



Hash function families

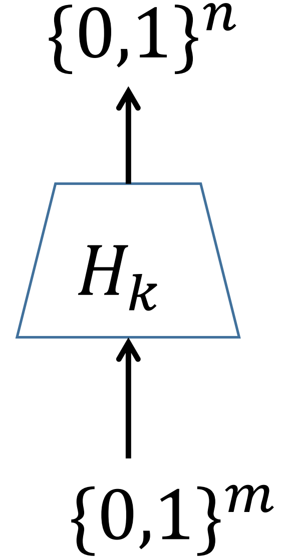
(Hash) function families
(aka. keyed functions)

$H: \{0,1\}^n \times \{0,1\}^m \rightarrow \{0,1\}^n$

$H_k(x) = H(k, x)$

Require $m \geq n$

and $H_k(x)$ is „efficient“



One-wayness

$H: \{0,1\}^n \times \{0,1\}^m \rightarrow \{0,1\}^n$

$k \leftarrow_R \{0,1\}^n$
$x \leftarrow_R \{0,1\}^m$
$y_c = H_k(x)$

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

y_c, k

$\mathcal{A}$ 

x^*

Collision resistance

$H: \{0,1\}^n \times \{0,1\}^m \rightarrow \{0,1\}^n$

$k \leftarrow_R \{0,1\}^n$

Success if

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

k

$\downarrow$

$\mathcal{A}$

$\downarrow$

(x_1^*, x_2^*)

Second-preimage resistance

$H: \{0,1\}^n \times \{0,1\}^m \rightarrow \{0,1\}^n$

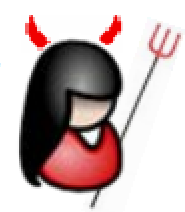
$k \leftarrow_R \{0,1\}^n$
$x_c \leftarrow_R \{0,1\}^m$

Success if

$H_k(x_c) = H_k(x^*)$ and
$x_c \neq x^*$

x_c, k

$\mathcal{A}$



x^*

Decisional version: Does a valid response exist?

Undetectability

$H: \{0,1\}^n \times \{0,1\}^m \rightarrow \{0,1\}^n$
$k \leftarrow_R \{0,1\}^n$
$b \leftarrow_R \{0,1\}$

If $b = 1$

$x \leftarrow_R \{0,1\}^m$

$y_c \leftarrow H_k(x)$

else

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

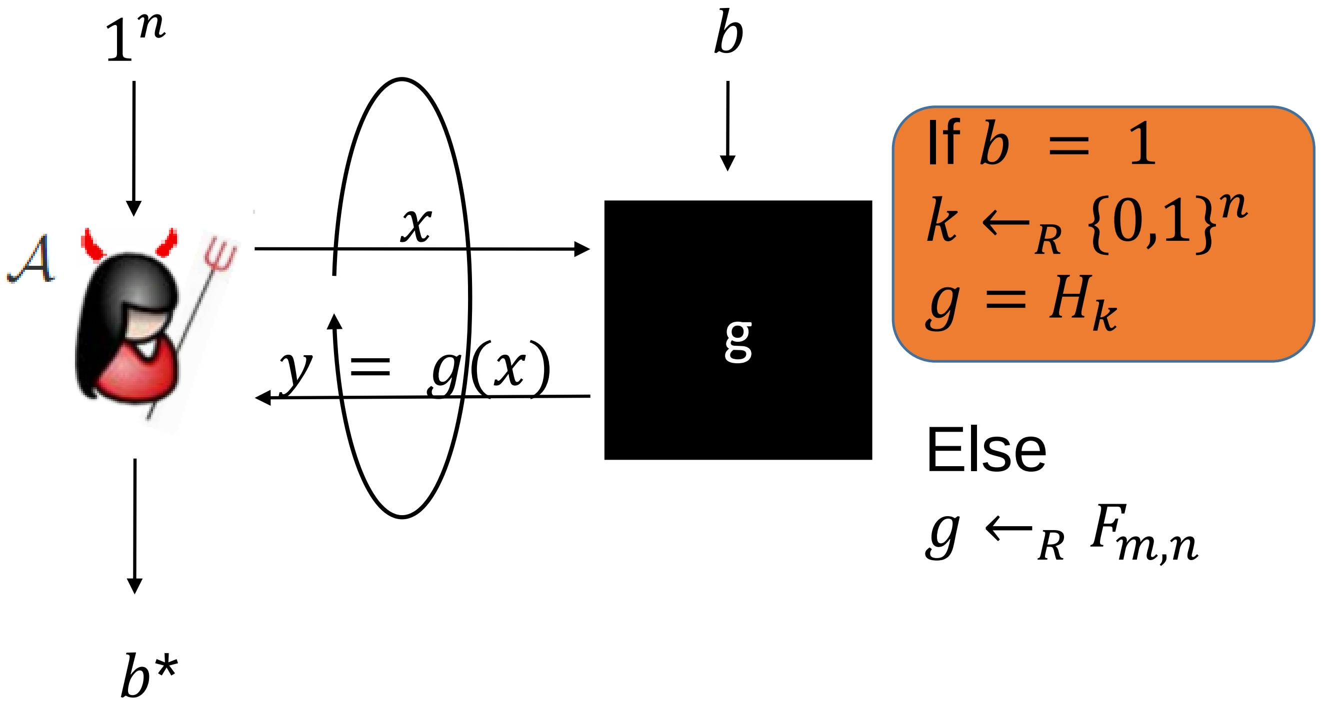
y_c, k

$\mathcal{A}$

$b*$

Pseudorandomness

$H\colon \{0,1\}^n \times \{0,1\}^m \rightarrow \{0,1\}^n$



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

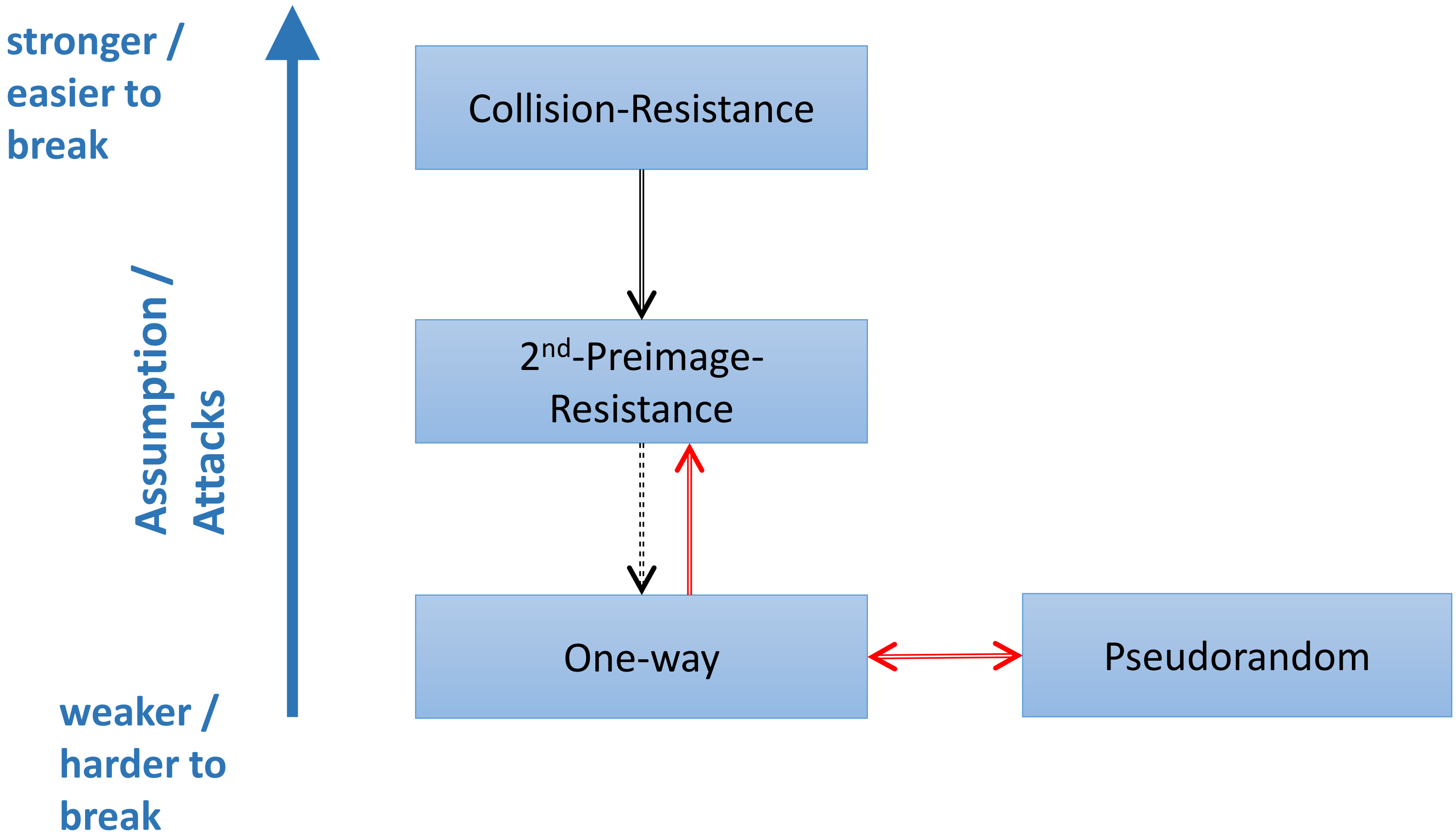
(Disclaimer: Currently only proof for $2^m \gg 2^n$)

Generic Security

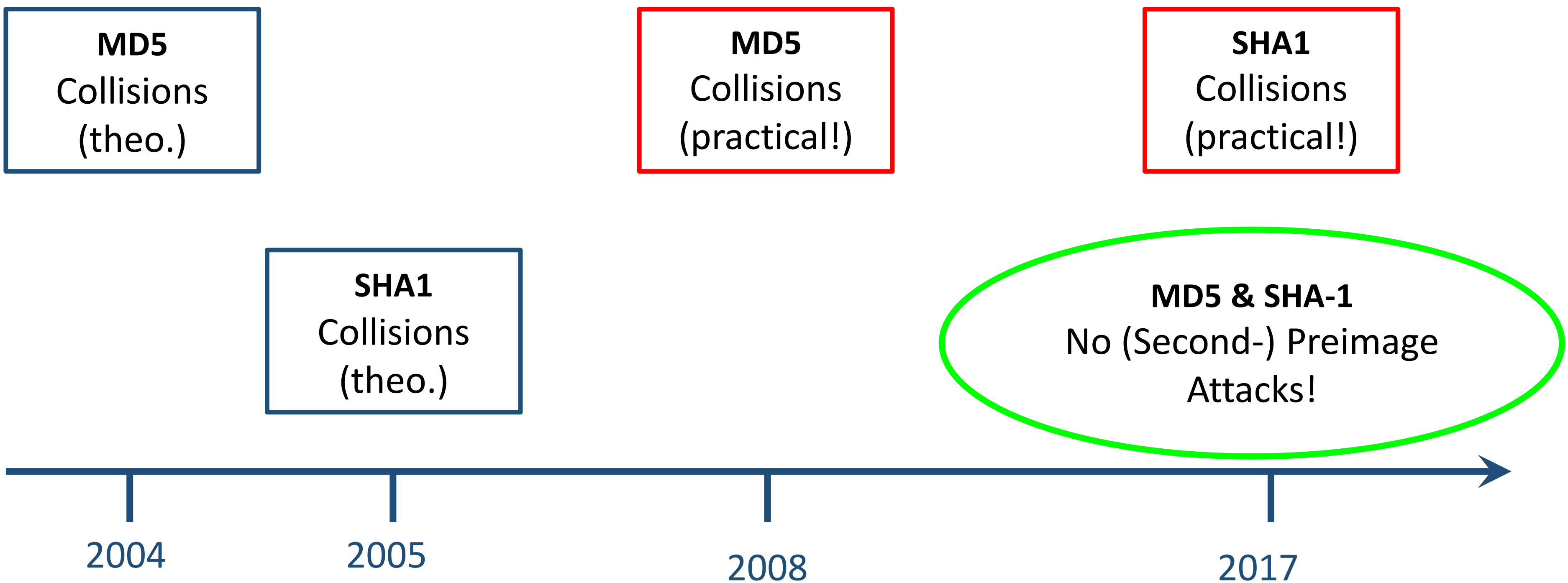
	OW	SPR	CR	UD*	PRF*
Classical	$\Theta(2^n)$	$\Theta(2^n)$	$\Theta(2^{n/2})$	$\Theta(2^n)$	$\Theta(2^n)$
Quantum	$\Theta(2^{n/2})$	$\Theta(2^{n/2})$	$\Theta(2^{n/3})$	$\Theta(2^{n/2})$	$\Theta(2^{n/2})$

* conjectured, no proof

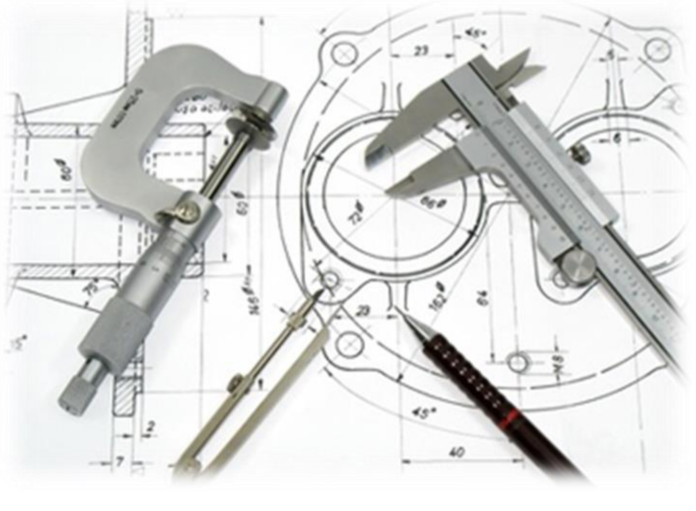
Hash-function properties



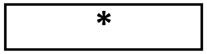
Attacks on Hash Functions

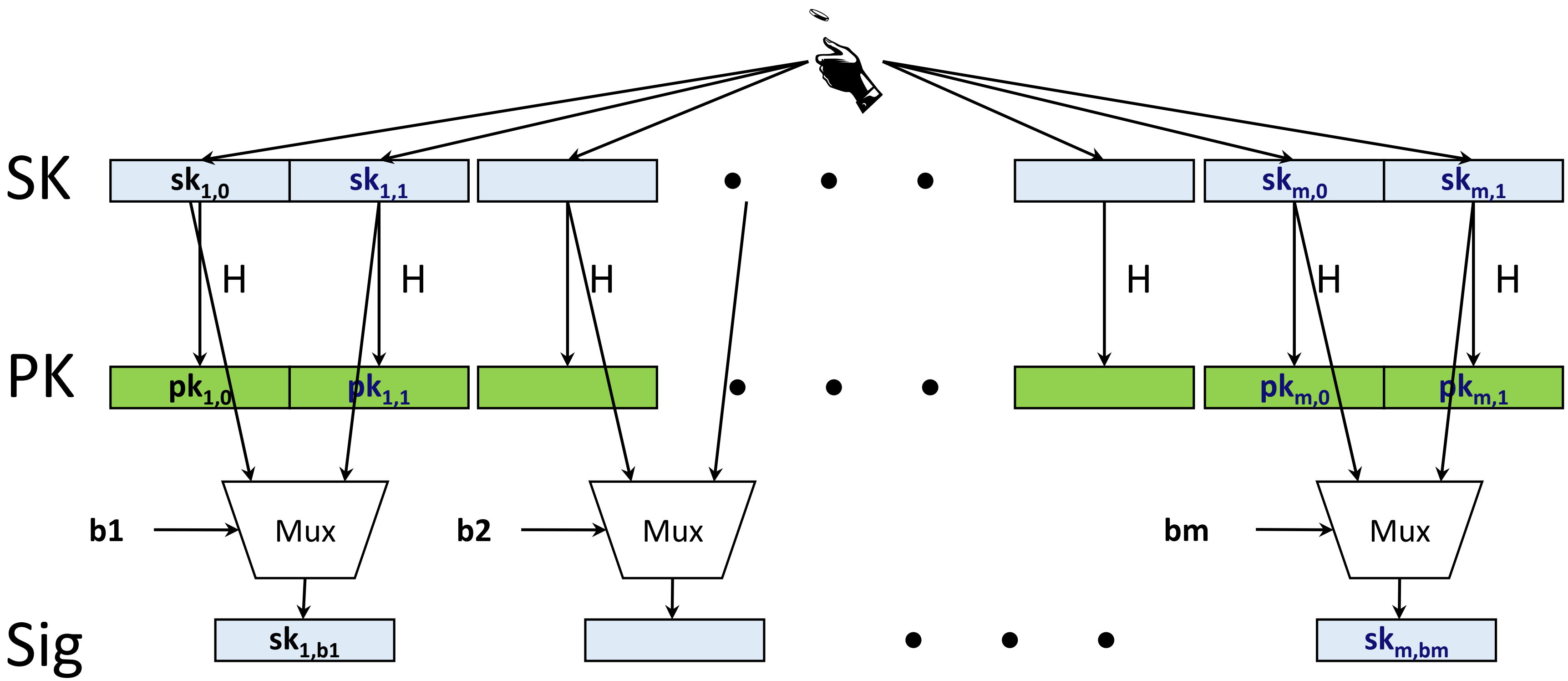


Basic Construction

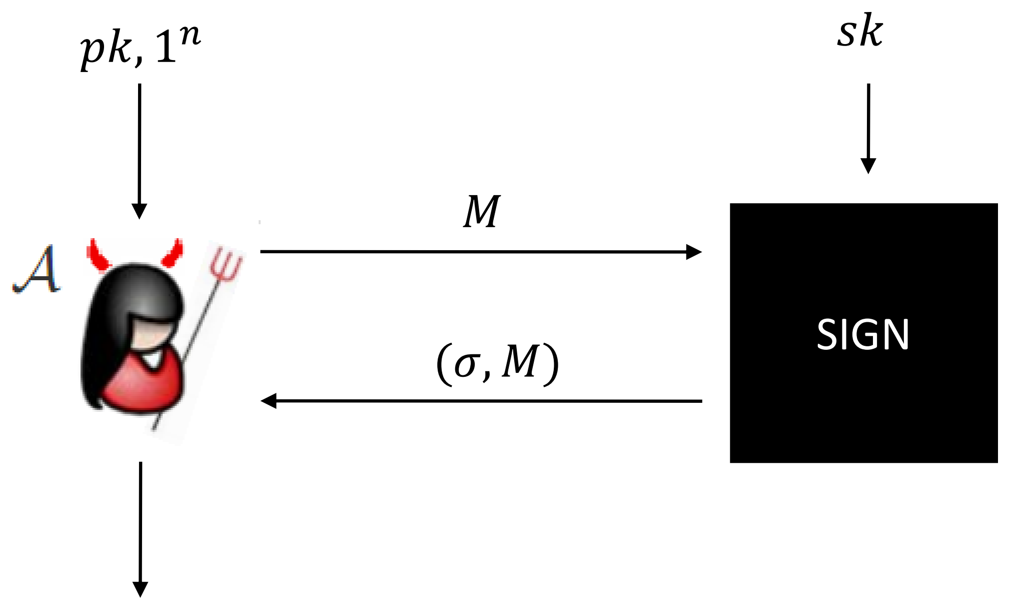


Lamport-Diffie OTS [Lam79]

Message M = b1,…,bm, OWF H    = n bit



EU-CMA for OTS



(σ^*, M^*)

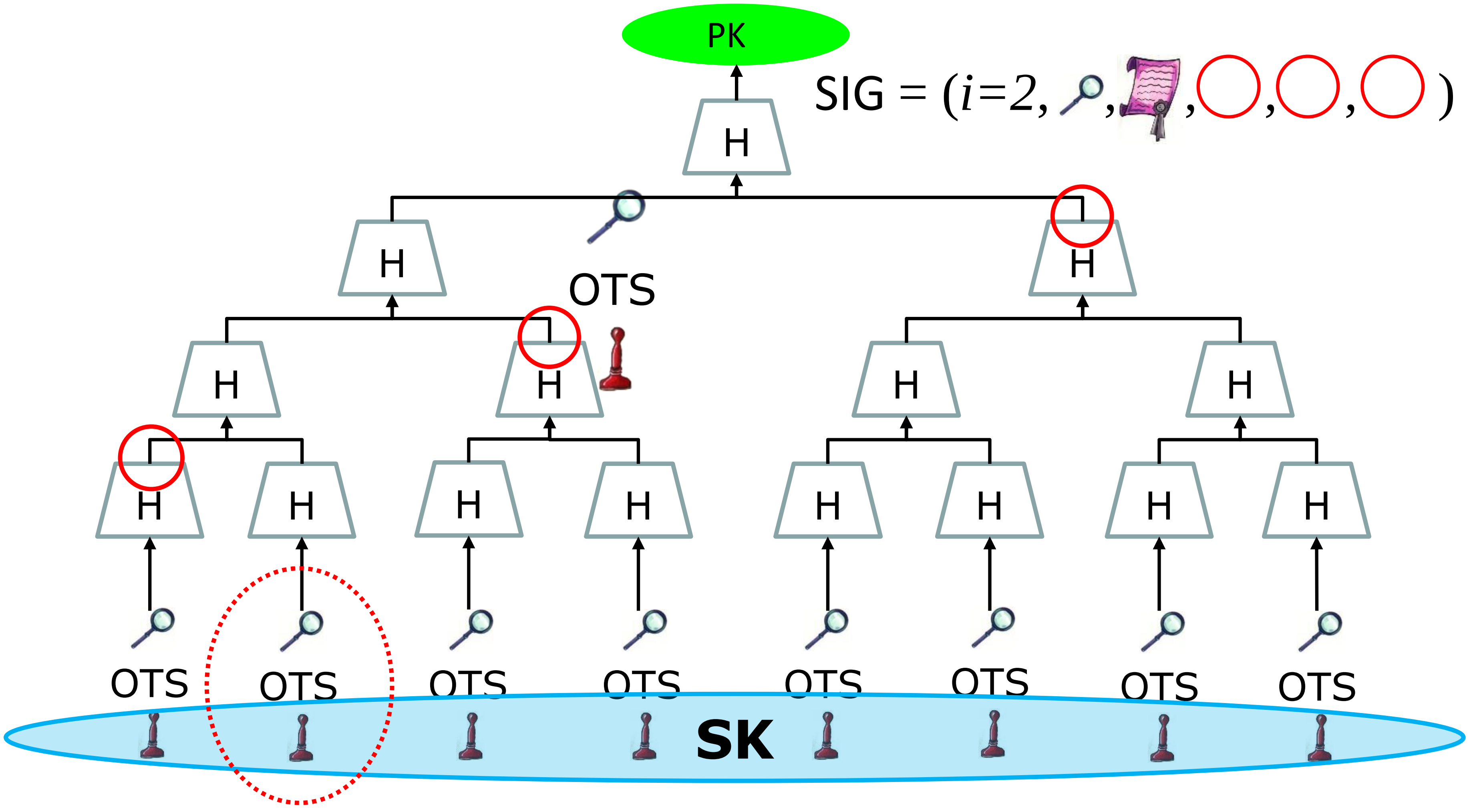
Success if $M^* \neq M$ and
Verify$(pk, \sigma^*, M^*) =$ Accept

Security

Theorem:

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

Merkle's Hash-based Signatures



Security

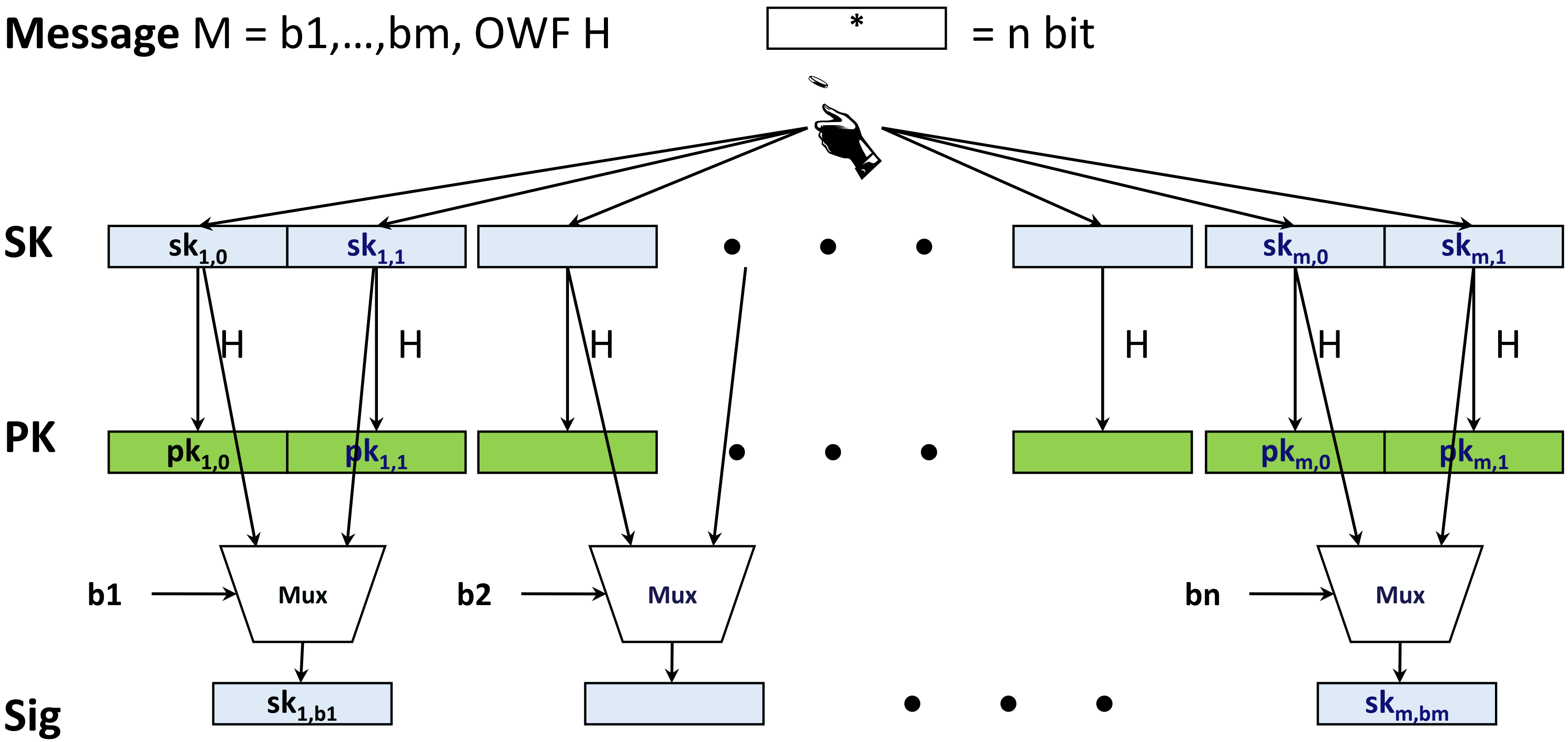
Theorem:

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.

Winternitz-OTS

Recap LD-OTS [Lam79]

Message M = b1,…,bm, OWF H



LD-OTS in MSS

SIG = (i=2, , , ◯, ◯, ◯)

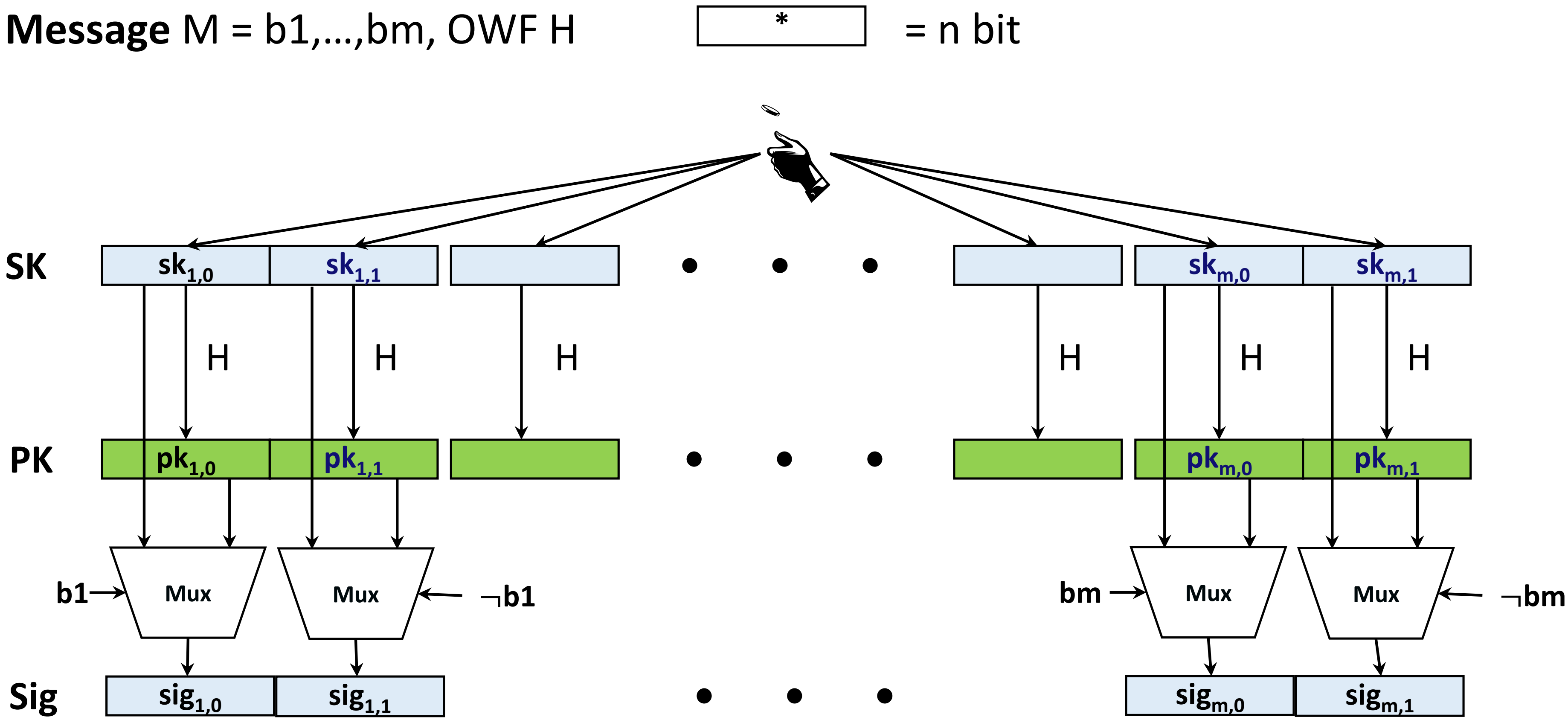
Verification:

1. Verify 
2. Verify authenticity of 

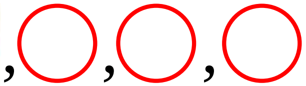
We can do better!

Trivial Optimization

Message M = b1,…,bm, OWF H



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 $b_1, \neg b_1,\ldots,b_m,\neg b_m$)

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

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: $sig_i =$ **=**

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

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

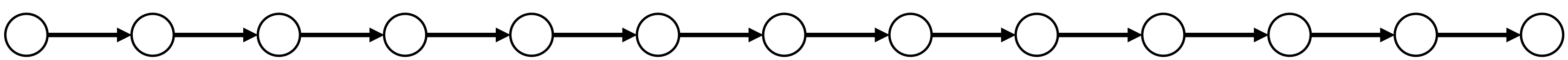
Function chains

Function family: $H: \{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}^n$
$k \leftarrow_R \{0,1\}^n$

Parameter w

Chain: $c^i(x) = H\left(c^{i-1}(x)\right) = \underbrace{H \circ H \circ \cdots \circ H}_{i\text{-times}}(x)$

$c^0(x) = x$

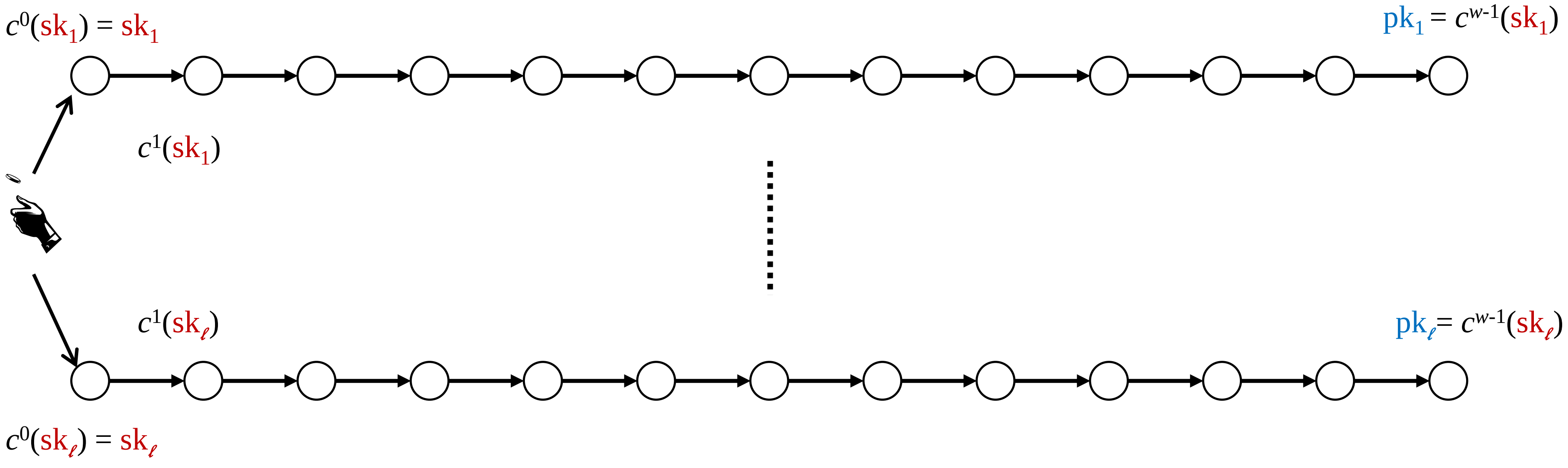


$c^1(x) = H_k(x)$ $\quad$ $\boldsymbol{c^{w-1}}(x)$

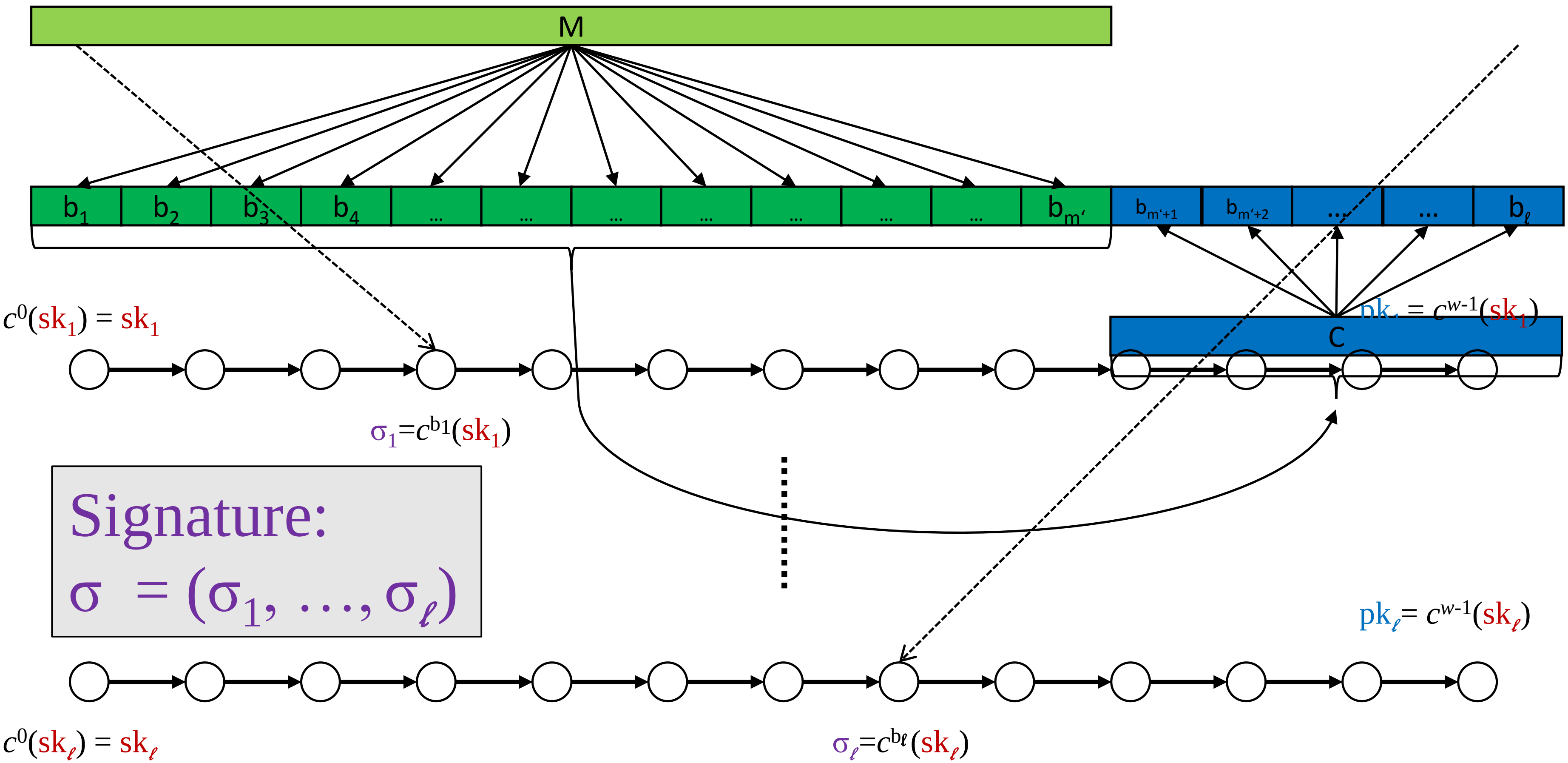
WOTS

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

Key Generation: Compute l, sample H_k

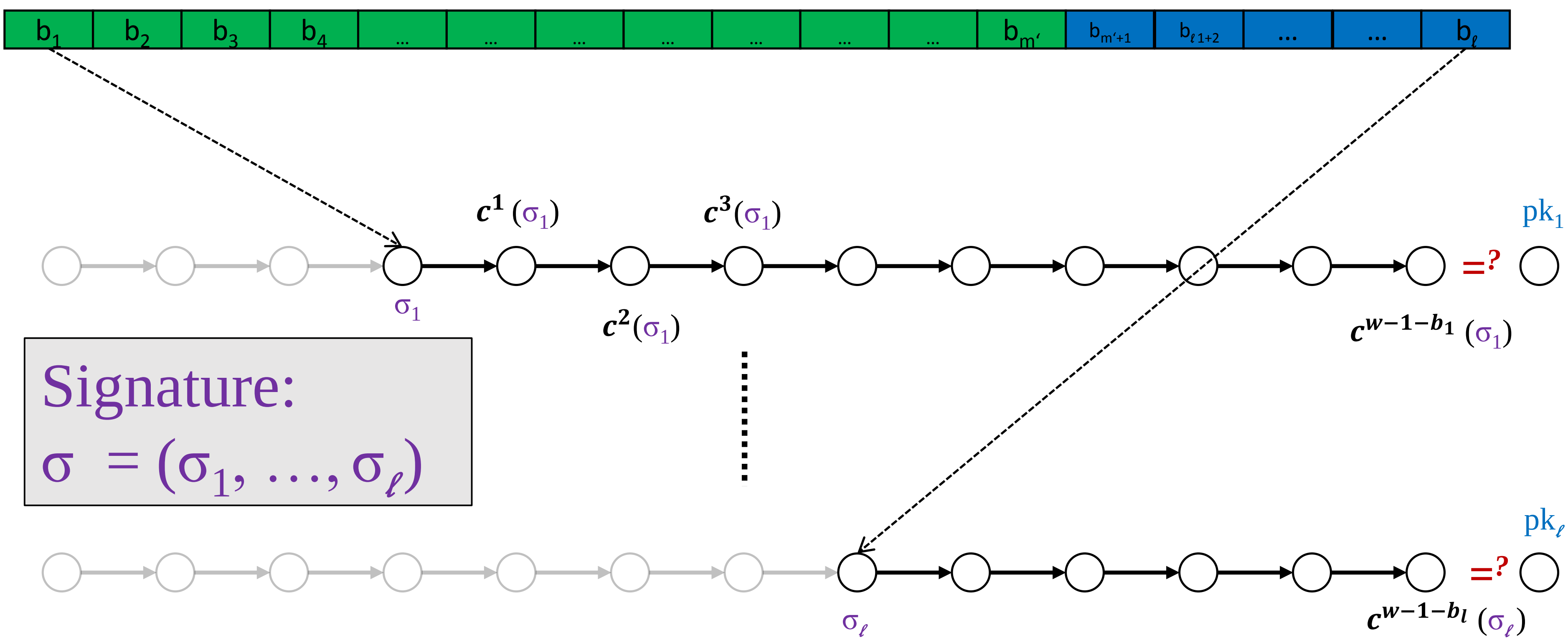


WOTS Signature generation



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$^+$: $c^i(x) = H_k(c^{i-1}(x) \oplus r_i)$

WOTS Security

Theorem (informally):

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

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

W-OTS⁺ *is strongly unforgeable under chosen message attacks if* H *is a* 2^{nd}*-preimage resistant and decisional* 2^{nd}*-preimage resistant family of functions.*

XMSS

XMSS

Tree: Uses bitmasks

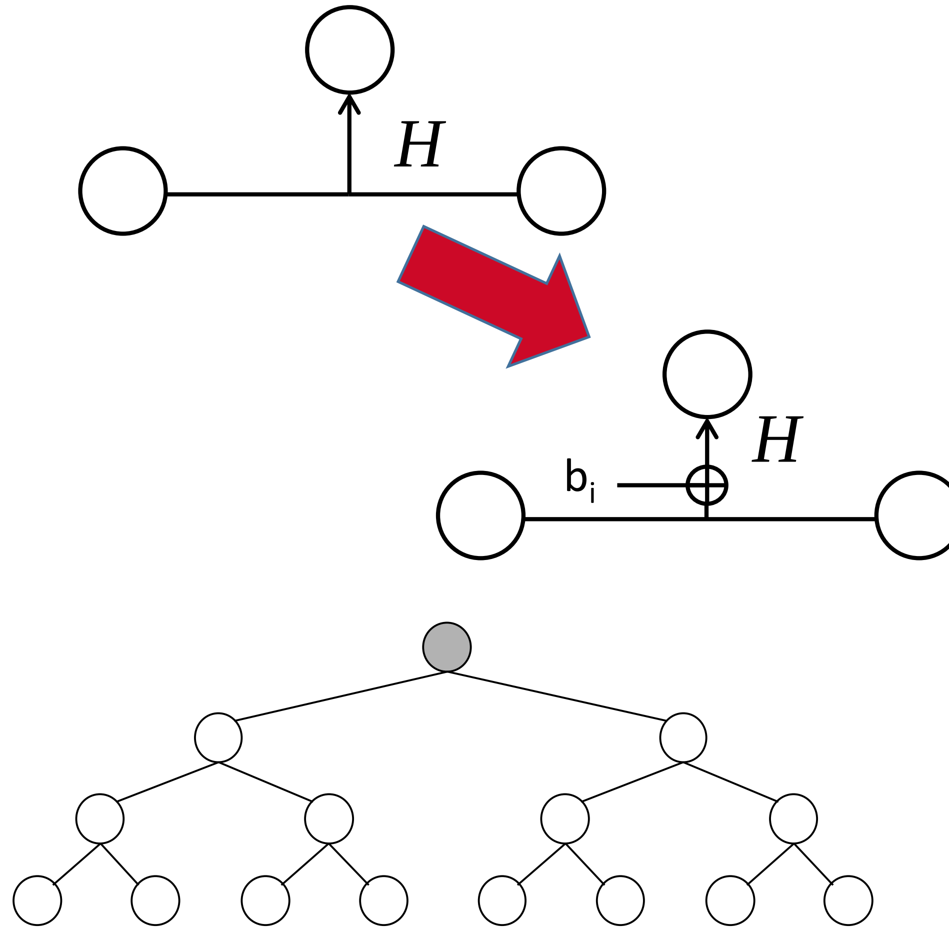
Leafs: Use binary tree with bitmasks

OTS: WOTS$^+$

Message digest: Randomized hashing

Collision-resilient

-> signature size halved

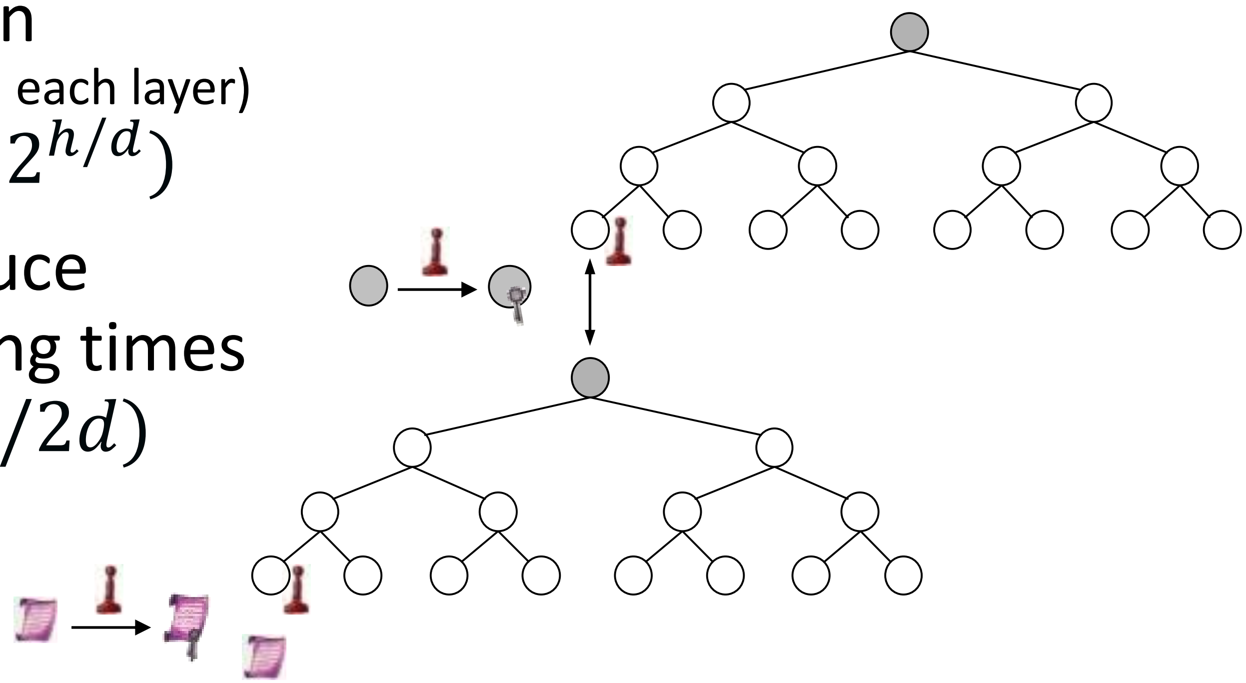


Multi-Tree XMSS

Uses multiple layers of trees

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

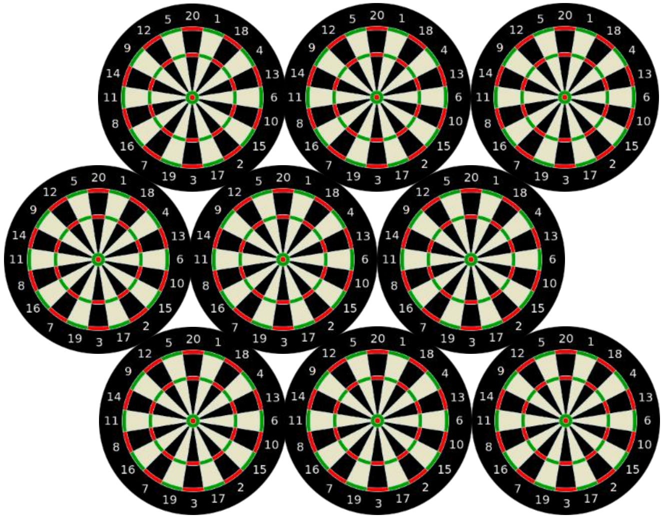
-> Allows to reduce worst-case signing times
$\Theta(h/2) \rightarrow \Theta(h/2d)$



Multi-target attacks

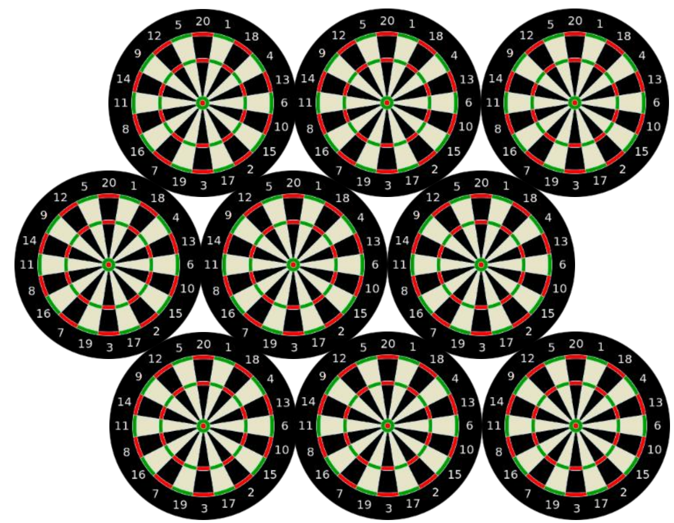
Multi-target attacks

- WOTS & Lamport need hash function h to be one-way
- Hypertree of total height 60 with WOTS (w=16) leads $> 2^{60} \cdot 67 \approx 2^{66}$ images.
- Inverting one of them allows existential forgery (at least massively reduces complexity)
- q-query brute-force succeeds with probability $\Theta\left(\frac{q}{2^{n-66}}\right)$ conventional and $\Theta\left(\frac{q^2}{2^{n-66}}\right)$ quantum
- We loose 66 bits of security! (33 bits quantum)



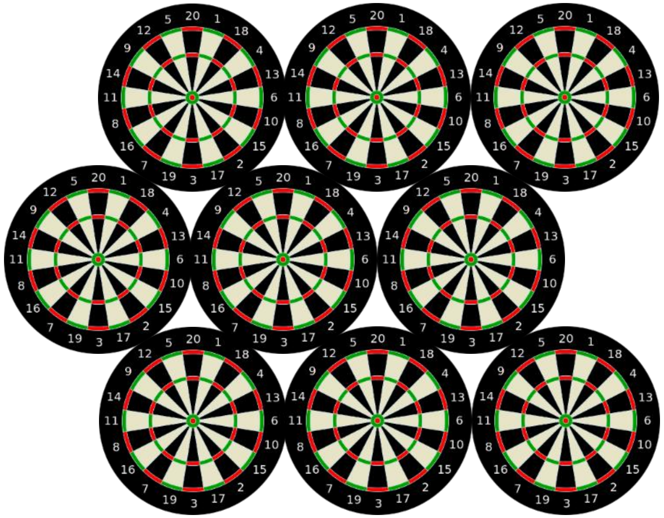
Multi-target attacks: Mitigation

- Mitigation: Separate targets [HRS16]
- Common approach:
  - In addition to hash function description
    and „input“ take
    - Hash „Address“
      (uniqueness in key pair)
    - Hash „key“ used for all hashes of one key pair
      (uniqueness among key pairs)



Multi-target attacks: Mitigation

- Mitigation: Separate targets [HRS16]
- Common approach:
  - In addition to hash function description
    and „input“ take
    - Hash „Address“
      (uniqueness in key pair)
    - Hash „key“ used for all hashes of one key pair
      (uniqueness among key pairs)



New intermediate abstraction: Tweakable Hash Function

- Tweakable Hash Function:

$$\mathbf{Th}(P, T, M) \rightarrow MD$$

P: Public parameters (one per key pair)
T: Tweak (one per hash call)
M: Message
MD: Message Digest

Security properties are determined by instantiation of tweakable hash!

XMSS in practice

[Docs] [txt|pdf] [draft-irtf-cfrg...] [Tracker] [Diff1] [Diff2] [Errata]

INFORMATIONAL
Errata Exist

Internet Research Task Force (IRTF)
Request for Comments: 8391
Category: Informational
ISSN: 2070-1721

A. Huelsing
TU Eindhoven
D. Butin
TU Darmstadt
S. Gazdag
genua GmbH
J. Rijneveld
Radboud University
A. Mohaisen
University of Central Florida
May 2018

XMSS: eXtended Merkle Signature Scheme

Abstract

This note describes the eXtended Merkle Signature Scheme (XMSS), a hash-based digital signature system that is based on existing descriptions in scientific literature. This note specifies Winternitz One-Time Signature Plus (WOTS+), a one-time signature scheme; XMSS, a single-tree scheme; and XMSS^MT, a multi-tree variant of XMSS. Both XMSS and XMSS^MT use WOTS+ as a main building block. XMSS provides cryptographic digital signatures without relying on the conjectured hardness of mathematical problems. Instead, it is proven that it only relies on the properties of cryptographic hash functions. XMSS provides strong security guarantees and is even secure when the collision resistance of the underlying hash function is broken. It is suitable for compact implementations, is relatively simple to implement, and naturally resists side-channel attacks. Unlike most other signature systems, hash-based signatures can so far withstand known attacks using quantum computers.

RFC 8391 -- XMSS: eXtended Merkle Signature Scheme

- Protecting against multi-target attacks / tight security
  - n-bit hash => n bit security
- Small public key (2n bit)
  - At the cost of (Q)ROM for proving PK compression secure
- Function families based on SHA2 & SHAKE (SHA3)

- 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

The LMS proposal

[Docs] [txt|pdf] [draft-mcgrew-ha...] [Tracker] [Diff1] [Diff2]

INFORMATIONAL

Internet Research Task Force (IRTF)
Request for Comments: 8554
Category: Informational
ISSN: 2070-1721

D. McGrew
M. Curcio
S. Fluhrer
Cisco Systems
April 2019

Leighton-Micali Hash-Based Signatures

Abstract

This note describes a digital-signature system based on cryptographic hash functions, following the seminal work in this area of Lamport, Diffie, Winternitz, and Merkle, as adapted by Leighton and Micali in 1995. It specifies a one-time signature scheme and a general signature scheme. These systems provide asymmetric authentication without using large integer mathematics and can achieve a high security level. They are suitable for compact implementations, are relatively simple to implement, and are naturally resistant to side-channel attacks. Unlike many other signature systems, hash-based signatures would still be secure even if it proves feasible for an attacker to build a quantum computer.

This document is a product of the Crypto Forum Research Group (CFRG) in the IRTF. This has been reviewed by many researchers, both in the research group and outside of it. The Acknowledgements section lists many of them.

Instantiating the tweakable hash (for SHA2)

XMSS

- K = SHA2(pad(PP)||TW),
  BM = SHA2(pad(PP)||TW+1),
  MD= SHA2(pad(K)||MSG ⊕ BM)
- Standard model proof if K & BM were random,
- (Q)ROM proof when generating K & BM as above (modeling those SHA2 invocations as RO)
- Tight proof is currently under revision

LMS

- MD = SHA2(PP||TW||MSG)
- QROM proof assuming SHA2 is QRO
- ROM proof assuming SHA2 compression function is RO
- Proofs are essentially tight

Instantiating the tweakable hash

- LMS is factor 3 faster but leads to slightly larger signatures at same security level
- LMS makes somewhat stronger assumptions about the security properties of the used hash function
- More research on direct constructions needed

SPHINCS

About the statefulness

- Works great for some settings

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

ELIMINATE THE STATE

Stateless hash-based signatures

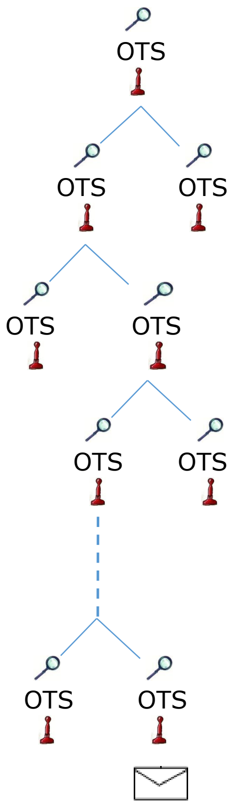
[NY89,Gol87,Gol04]

Goldreich's approach [Gol04]:

Security parameter $\lambda = 128$

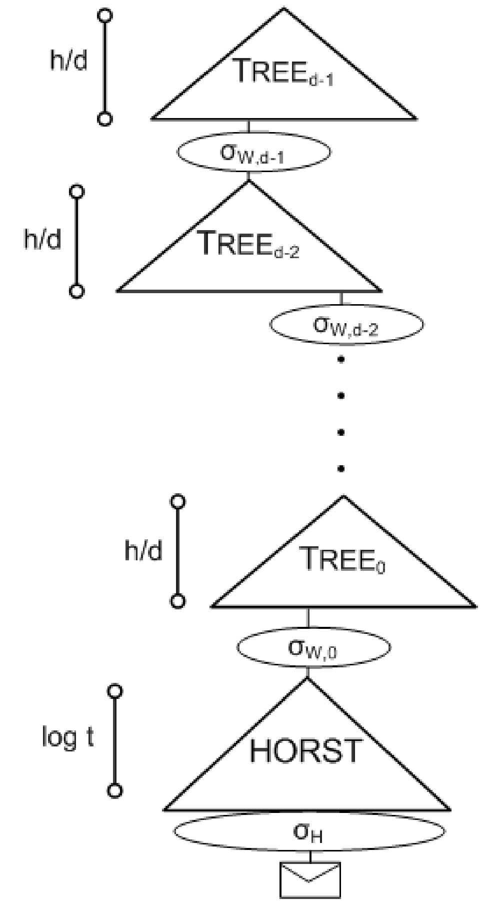
Use binary tree as in Merkle, but...

- ...for security
  - pick index i at random;
  - requires huge tree to avoid index collisions (e.g., height $\mathrm{h} = 2\lambda = 256$).
- ...for efficiency:
  - use binary certification tree of OTS key pairs (= Hypertree with $d = h$),
  - all OTS secret keys are generated pseudorandomly.



SPHINCS [BHH+15]

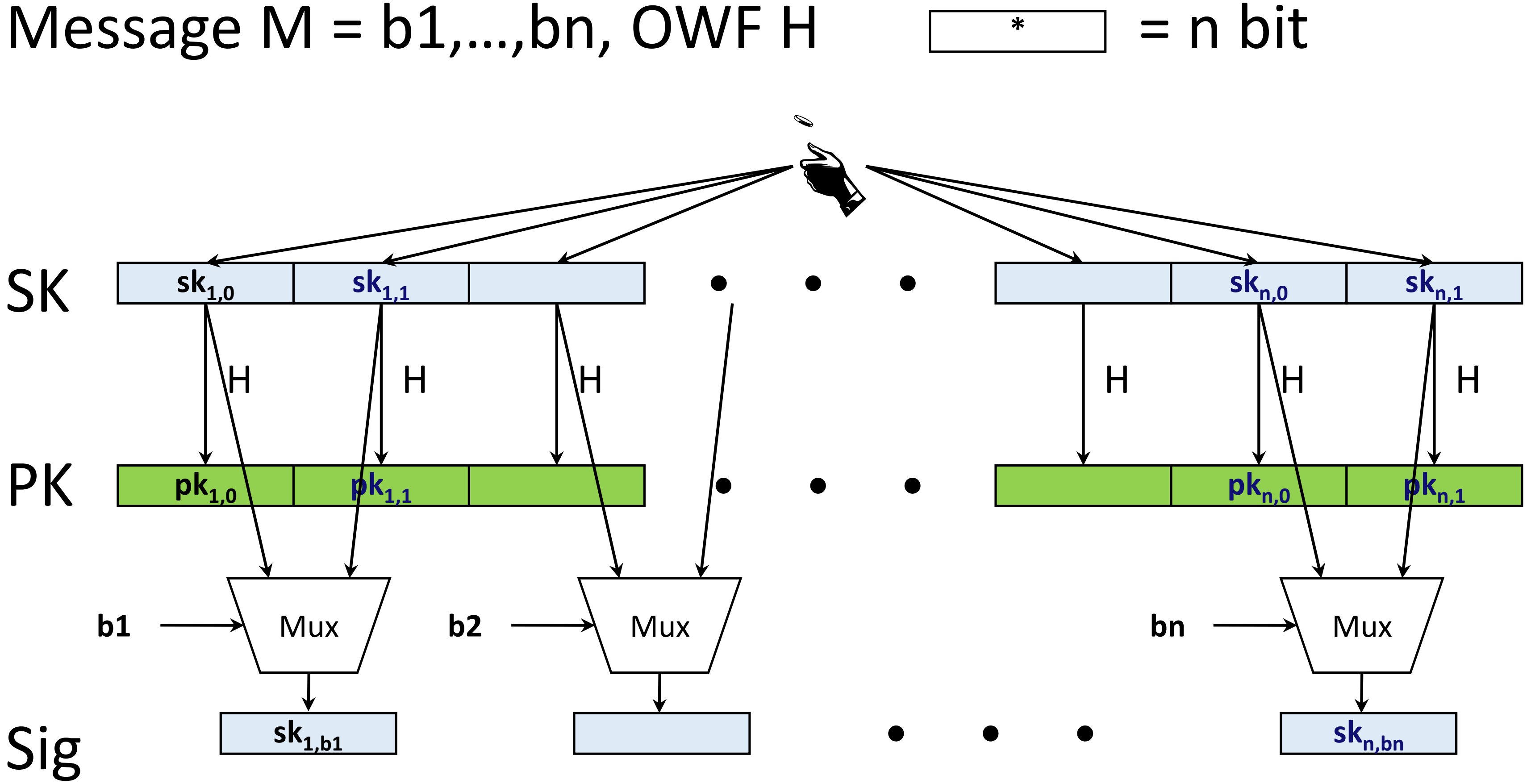
- Select index pseudo-randomly
- Use a few-time signature key-pair on leaves to sign messages
  - Few index collisions allowed
  - Allows to reduce tree height
- Use hypertree: Use d << h.



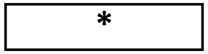
Few-Time Signature Schemes

Recap LD-OTS

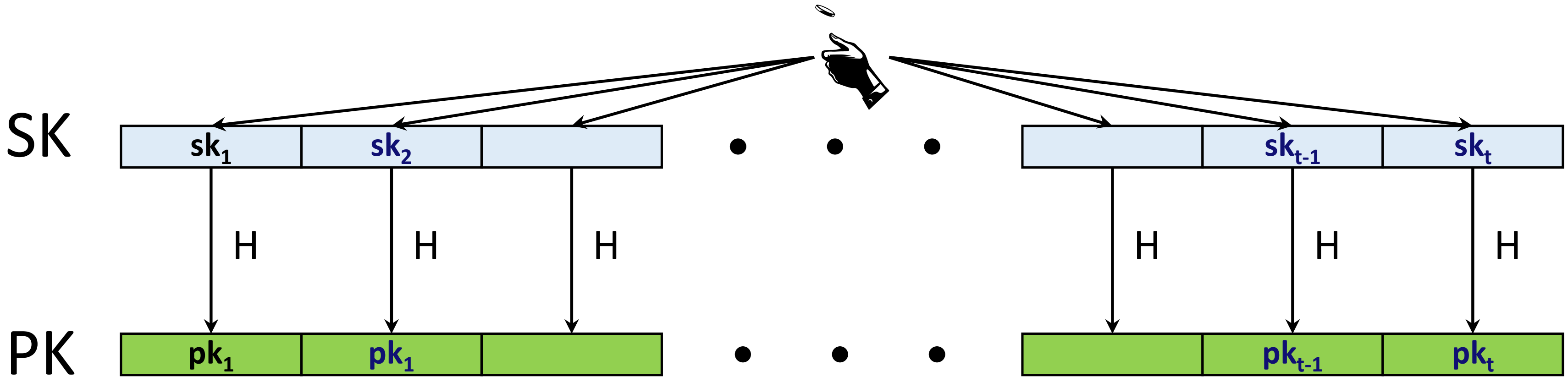
Message M = b1,…,bn, OWF H



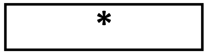
HORS [RR02]

Message M, OWF H, CRHF H’  = n bit

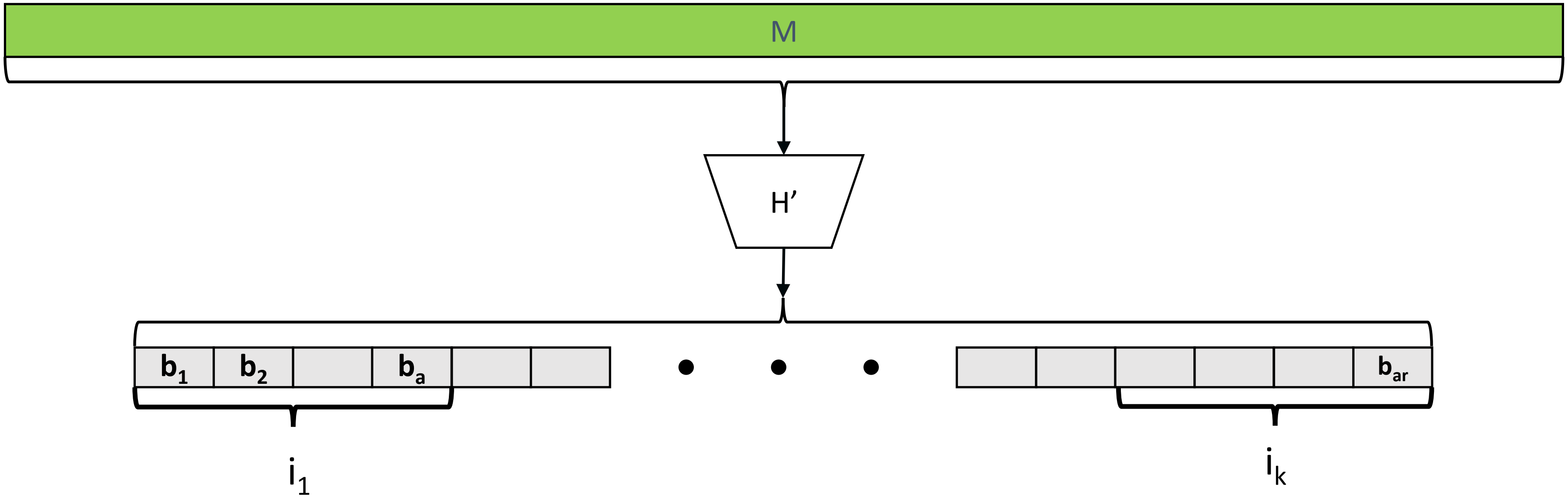
Parameters $t=2^a$,k, with m = ka (typical a=16, k=32)



HORS mapping function

Message M, OWF H, CRHF H’  = n bit

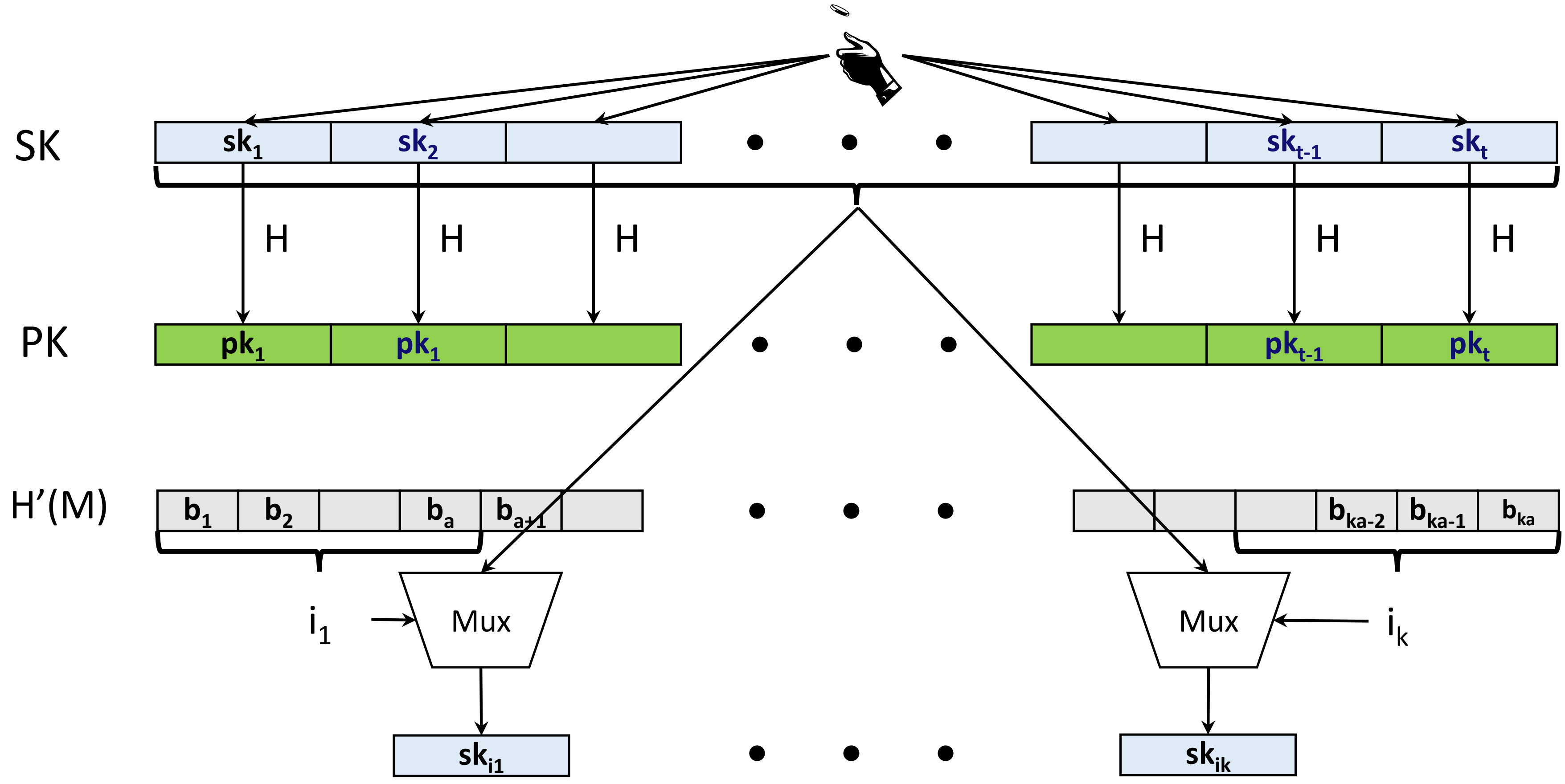
Parameters $t=2^a$,k, with m = ka (typical a=16, k=32)



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 $msg_j, 1 \le j \le r$, hard to find $msg_{r+1} \neq msg_j$ such that $M_{r+1}^i \in \bigcup_{1 \le j \le r} M_j^i$.

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

$\longrightarrow$ Security shrinks with each signature!

IF YOU LIKE IT

YOU SHOULD

PUT A

TREE

~~RING~~ ON IT

HORST

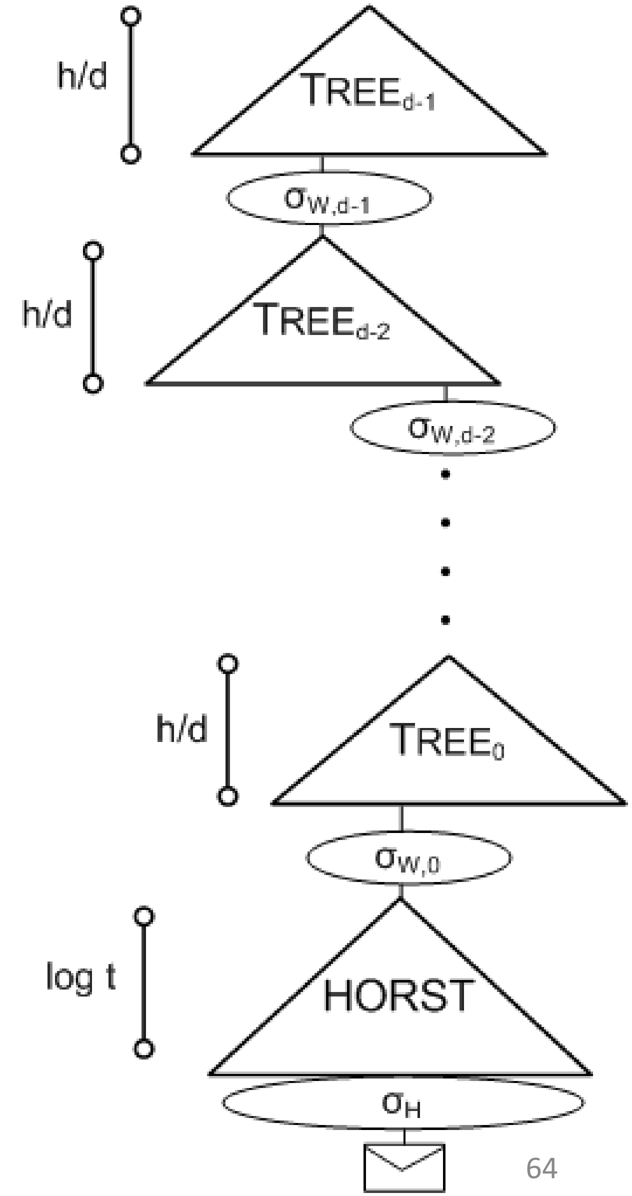
Using HORS with MSS requires adding PK (tn) to MSS signature.

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 $\rightarrow$ 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



SPHINCS$^+$

Joint work with Jean-Philippe Aumasson, Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe

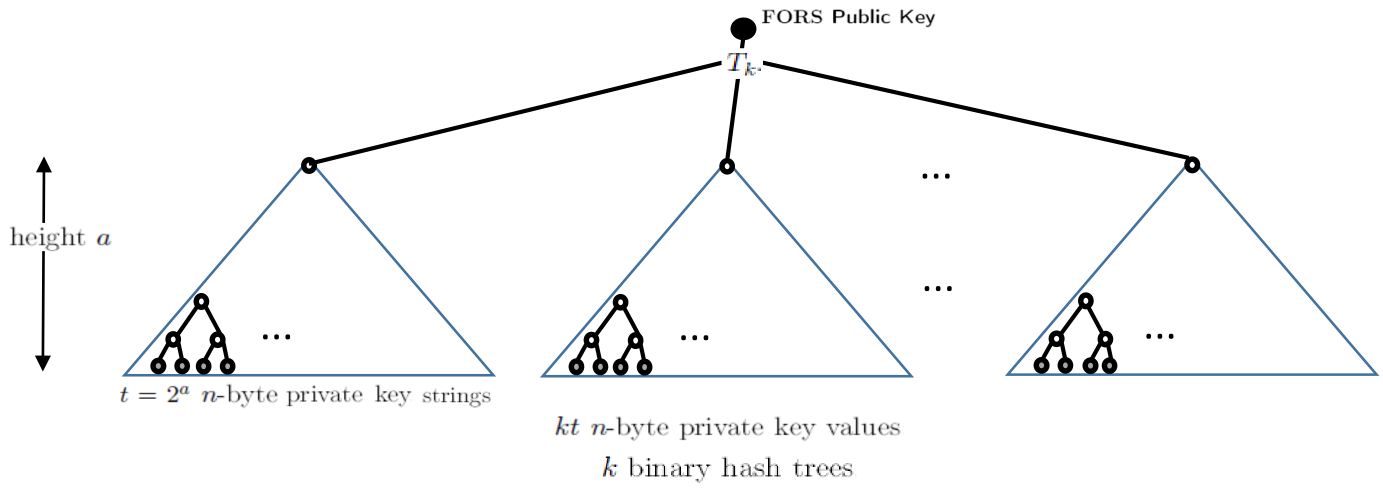
SPHINCS$^+$ (our NIST submission)

- Strengthened security gives smaller signatures
- Collision- and multi-target attack resilient
- Fixed length signatures
- Small keys, medium size signatures (lv 3: 17kB)
- Sizes can be much smaller if q_sign gets reduced
- The **conservative choice**


PUT A
TREES
RING ON IT

FORS (Forest of random subsets)

- Parameters t, a = log t, k such that ka = m



https://huelsing.net

Verifiable index selection

(and optionally non-deterministic randomness)

- SPHINCS:

$$(\text{idx}||\mathbf{R}) = PRF(\mathbf{SK}.\text{prf}, M)$$
$$\text{md} = H_{\text{msg}}(\mathbf{R}, \text{PK}, M)$$

- SPHINCS$^+$:

$$\mathbf{R} = PRF(\mathbf{SK}.\text{prf}, \text{OptRand}, M)$$
$$(\text{md}||\text{idx}) = H_{\text{msg}}(\mathbf{R}, \text{PK}, M)$$

Verifiable index selection

Improves FORS security

- SPHINCS:
  Attacks can target „weakest“ HORST key pair
- SPHINCS$^+$:
  Every hash query also selects FORS key pair
  - Leads to notion of interleaved target subset resilience

Instantiations (after second round tweaks)

- SPHINCS$^+$-SHAKE256-robust
- SPHINCS$^+$-SHAKE256-simple NEW!
- SPHINCS$^+$-SHA-256-robust
- SPHINCS$^+$-SHA-256-simple NEW!
- SPHINCS$^+$-Haraka-robust
- SPHINCS$^+$-Haraka-simple NEW!

Instantiations (small vs fast)

	n	h	d	$\log(t)$	k	w	bitsec	sec level	sig bytes
SPHINCS$^+$-128s	16	64	8	15	10	16	133	**1**	8 080
SPHINCS$^+$-128f	16	60	20	9	30	16	128	**1**	16 976
SPHINCS$^+$-192s	24	64	8	16	14	16	196	**3**	17 064
SPHINCS$^+$-192f	24	66	22	8	33	16	194	**3**	35 664
SPHINCS$^+$-256s	32	64	8	14	22	16	255	**5**	29 792
SPHINCS$^+$-256f	32	68	17	10	30	16	254	**5**	49 216

Hash-based Signatures in NIST „Competition“

- SPHINCS$^+$
  - FORS as few-time signature
  - XMSS-T tweakable hash
- Gravity-SPHINCS (R.I.P.)
  - PORS as few-time signature
  - Requires collision-resistance
  - Vulnerable to multi-target attacks
- (PICNIC)

Table 2: Performance comparison of different symmetric-crypto-based signature schemes on the Intel Haswell microarchitecture. All software is optimized using architecture-specific optimizations such as AESNI or AVX2 instructions.

Scheme	Cycles keypair	Cycles sign	Cycles verify	Bytes sig	Bytes pk	Bytes sk
Comparison to SPHINCS-256						
SPHINCS-256 [8]	2 868 464[a]	50 462 856[a]	1 672 652[a]	41 000	1 056	1 088
SPHINCS$^+$ (Haraka, robust) ($n = 192, h = 51, d = 17, b = 7, k = 45, w = 16$)	1 254 968[b]	29 015 002[b]	2 739 770[b]	30 696	48	96
Comparison to Gravity-SPHINCS						
Gravity-SPHINCS [5] (parameter-set L)	30 729 044 392[a]	32 564 796[a]	625 752[a]	max: 35 168 avg: ?[c]	32	1 048 608
SPHINCS$^+$ (Haraka robust) ($n = 192, h = 66, d = 22, b = 8, k = 33, w = 16$)	1 257 826[b]	38 840 268[b]	3 467 192[b]	35 664	48	96
SPHINCS$^+$ (Haraka, simple) ($n = 192, h = 64, d = 16, b = 7, k = 49, w = 16$)	1 892 462[b]	35 029 380[b]	1 460 204[b]	30 552	48	96
Comparison to Picnic						
Picnic2-L5-FS [15]	35 716[c]	1 346 724 260[c]	387 637 876[c]	max: 54 732 avg: 46 282	65	97
SPHINCS$^+$ (SHA-256, simple) ($n = 256, h = 64, d = 8, b = 14, k = 22, w = 16$)	85 946 882[b]	1 121 074 298[b]	4 903 926[b]	29 792	64	128

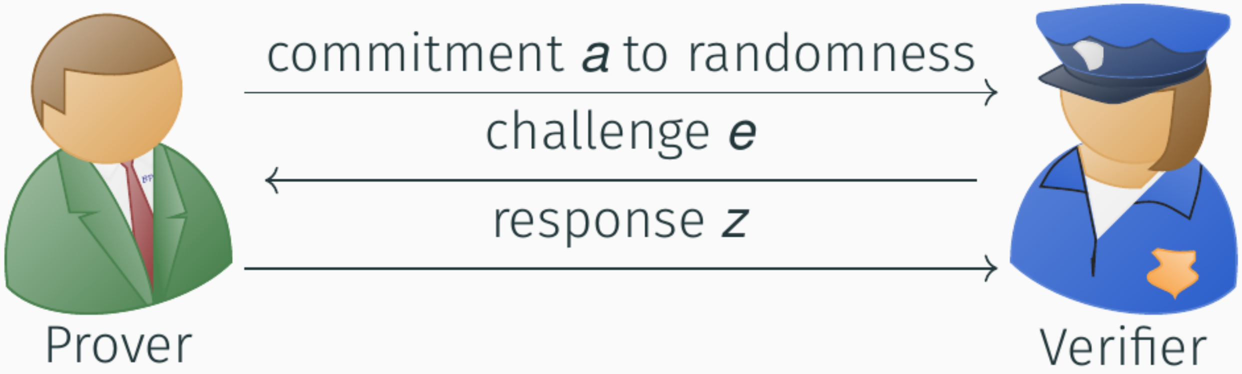
[a] As reported by SUPERCOP [10] from 3.5GHz Intel Xeon E3-1275 V3 (Haswell)
[b] Median of 100 runs on 3.5GHz Intel Xeon E3-1275 V3 (Haswell), compiled with `gcc-5.4 -O3 -march=native -fomit-frame-pointer -flto`
[c] As reported by [15] for the optimized implementation on a 3.6GHz Intel Core i7-4790K (Haswell)
[d] Neither [5] nor [6] report the average size of signatures; the analysis in [4] suggests that it is about 1KB smaller than the worst-case size.

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

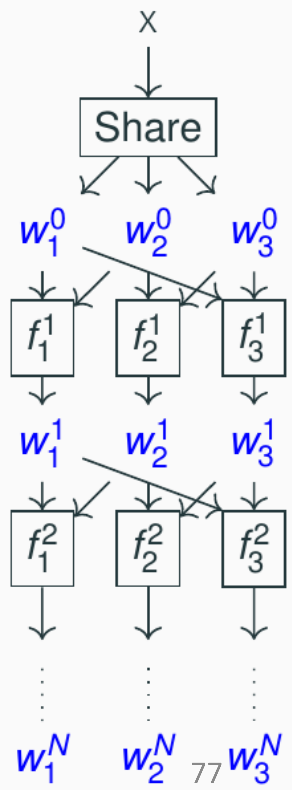
ZKBoo

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

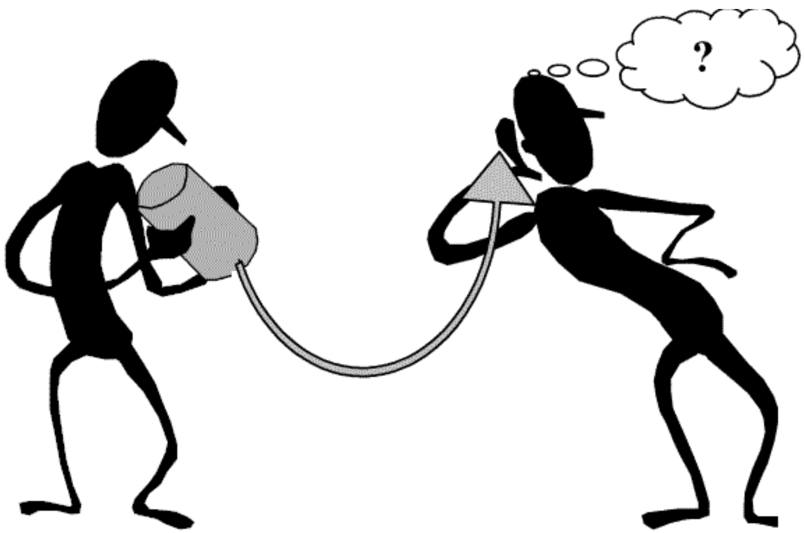
High-Level Approach

- Use LowMC v2 to build dedicated hash function with low #AND-gates
- 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)

Conclusion

- If you can live with a state, you have PQ signatures available with XMSS & LMS
- For stateless we are waiting for NIST to finish: SPHINCS+ & Picnic in second round

Thank you!
Questions?



For references & further literature see
https://huelsing.net/wordpress/?page_id=165

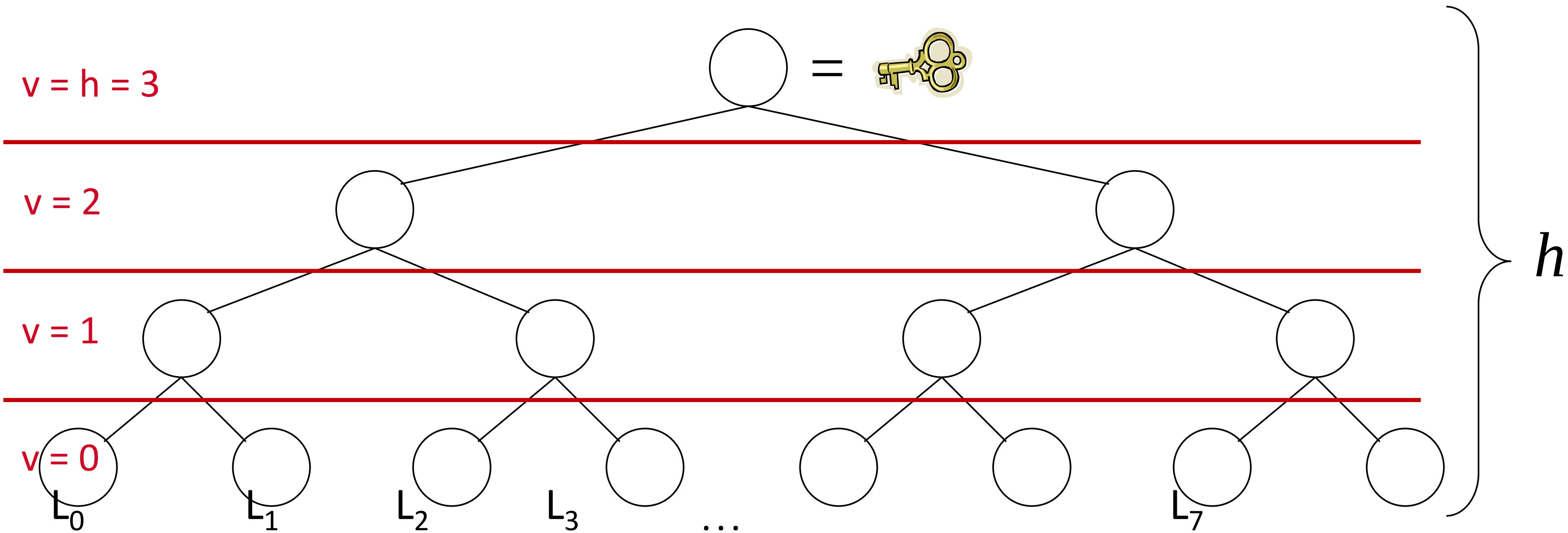
Authentication path computation

TreeHash

(Mer89)

TreeHash

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



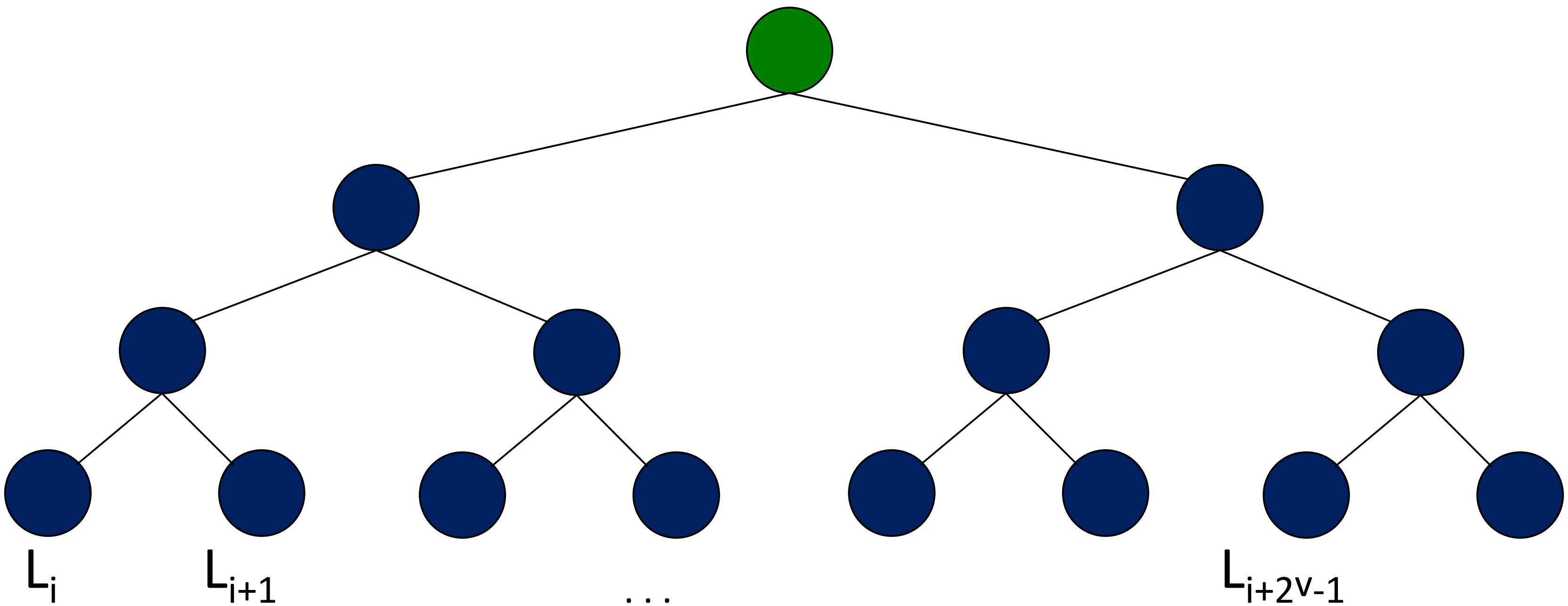
TreeHash

TreeHash(v,i)

```
Init Stack, N1, N2
For j = i to i+2^v-1 do
        N1 = LeafCalc(j)
        While N1.level() == Stack.top().level() do
                N2 = Stack.pop()
                N1 = ComputeParent( N2, N1 )
        Stack.push(N1)
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

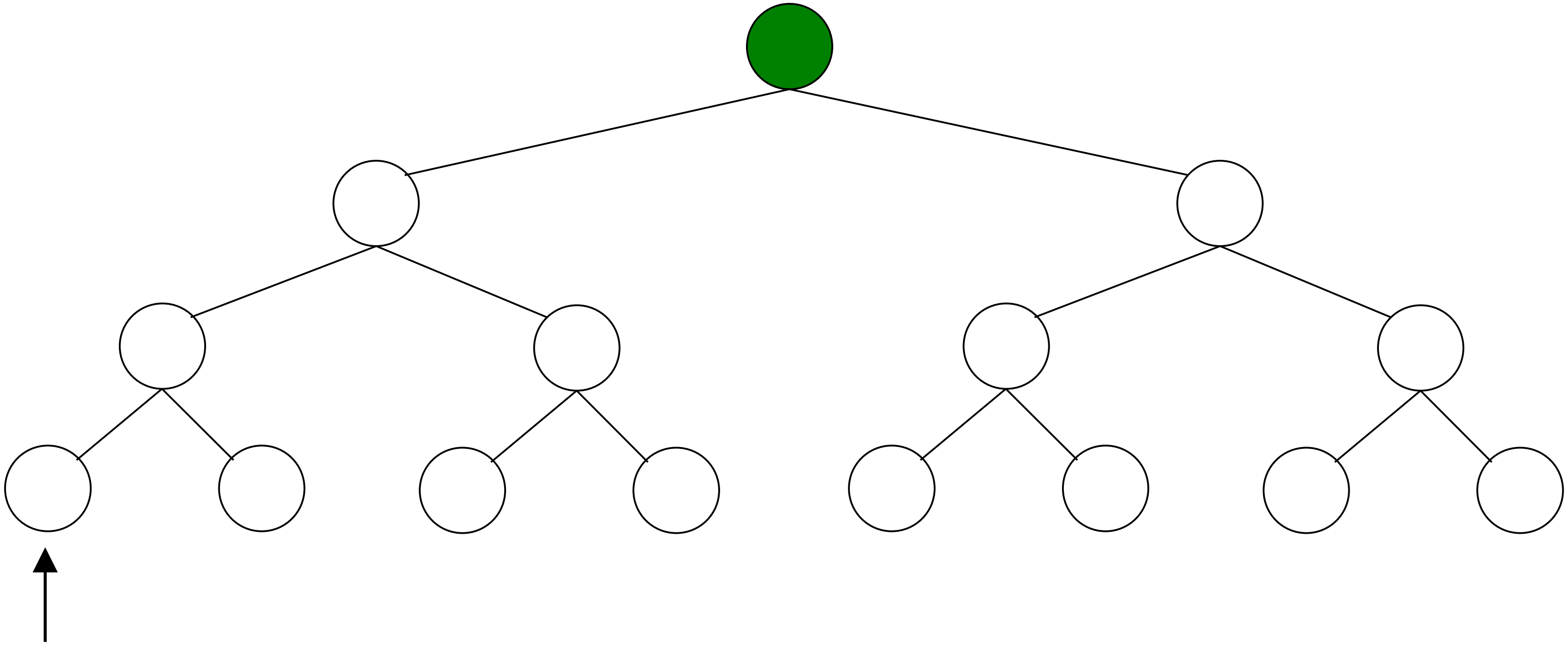
Store whole tree: $2^h n$ bits.

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)

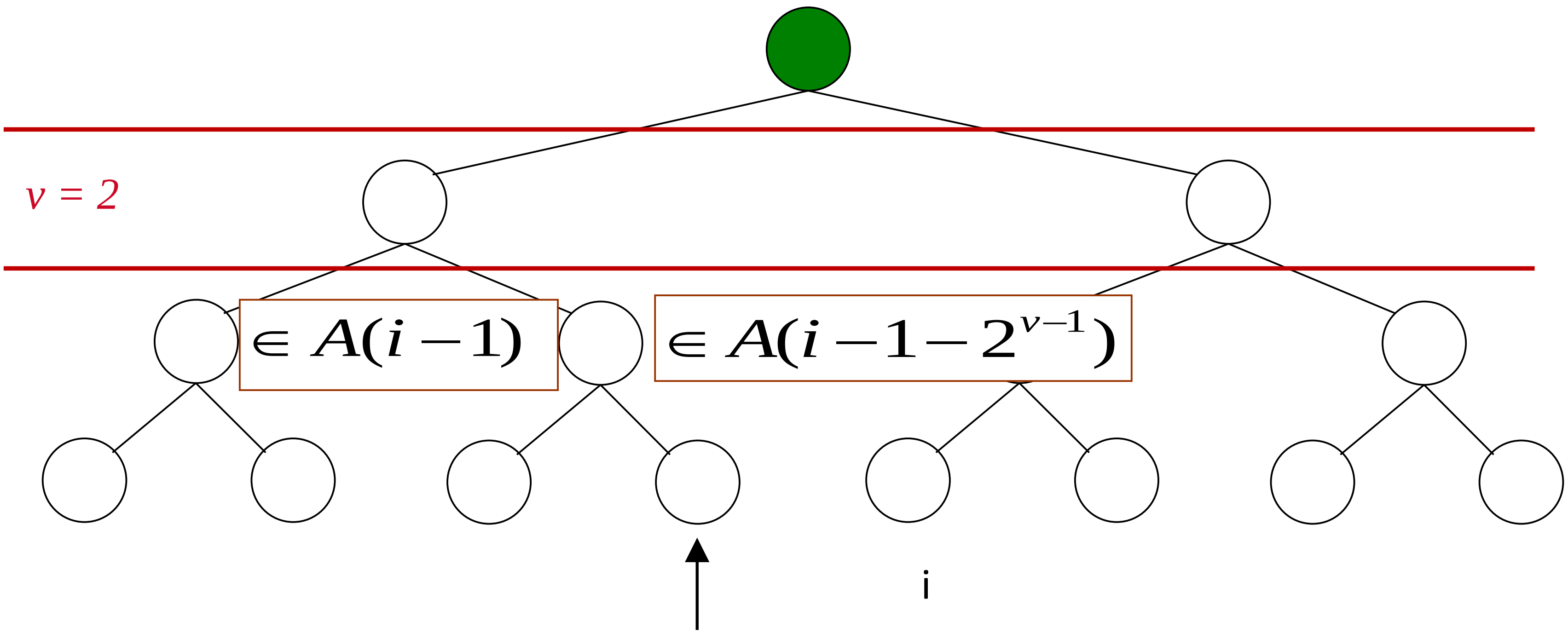
But: Worst case still 2^h-1 leaf + 2^h-1-h node computations!

-> 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:

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

Runtime:

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

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