Hash-based signatures

Tanja Lange
(with some slides by Daniel J. Bernstein)

Eindhoven University of Technology

20 July 2020

Benefits of hash-based signatures

- Old idea: 1979 Lamport one-time signatures.
- 1979 Merkle extends to more signatures;
  many further improvements in years since.
- Security thoroughly analyzed.
- Only one prerequisite: a good hash function, e.g. SHA3-512, ...
  Hash functions map long strings to fixed-length strings.

$$H : \{0,1\}^* \to \{0,1\}^n.$$

  Signature schemes use hash functions in handling m.
- Cryptographic hash functions are computationally
  - preimage resistant: function is one way;
  - second preimage resistant:
    given $x, H(x)$ cannot find $x' \neq x$ with $H(x') = H(x)$;
  - collision resistant: cannot find $x' \neq x$ with $H(x') = H(x)$.

  Quantum computers affect the hardness only marginally
  finding preimages in $2^{n/2}$ instead of 2^n (Grover, not Shor).

A signature scheme for empty messages: key generation

A signature scheme for empty messages: key generation

First part of signempty.py

```
import os; import hashlib;

def keypair():
  secret = sha3_256(os.urandom(32))
  public = sha3_256(secret)
  return public,secret
```

A signature scheme for empty messages: key generation

First part of signempty.py

```
import os; import hashlib;

def keypair():
  secret = sha3_256(os.urandom(32))
  public = sha3_256(secret)
  return public,secret
```

```
>>> import signempty; import binascii;
>>> pk,sk = signempty.keypair()
>>> binascii.hexlify(pk)
b'a447bc8d7c661f85defcf1bbf8bad77bfc6191068a8b658c99c7ef4cbe37cf
>>> binascii.hexlify(sk)
b'a4a1334a6926d04c4aa7cd98231f4b644be90303e4090c358f2946f1c25768
```

A signature scheme for empty messages: signing, verification

Rest of signempty.py

```
def sign(message,secret):
  if message != '': raise Exception('nonempty message')
  signedmessage = secret
  return signedmessage

def open(signedmessage,public):
  if sha3_256(signedmessage) != public:
    raise Exception('bad signature')
  message = ''
  return message
```

A signature scheme for empty messages: signing, verification

Rest of signempty.py

```
def sign(message,secret):
  if message != '': raise Exception('nonempty message')
  signedmessage = secret
  return signedmessage

def open(signedmessage,public):
  if sha3_256(signedmessage) != public:
    raise Exception('bad signature')
  message = ''
  return message
```

```
>>> sm = signempty.sign('',sk)
>>> signempty.open(sm,pk)
''
```

A signature scheme for 1-bit messages: key generation, signing

A signature scheme for 1-bit messages: key generation, signing

First part of signbit.py

```
import signempty

def keypair():
  p0,s0 = signempty.keypair()
  p1,s1 = signempty.keypair()
  return p0+p1,s0+s1

def sign(message,secret):
  if message == 0:
    return ('0' , signempty.sign('',secret[0:32]))
  if message == 1:
    return ('1' , signempty.sign('',secret[32:64]))
  raise Exception('message must be 0 or 1')
```

A signature scheme for 1-bit messages: verification

Rest of signbit.py

```
def open(signedmessage,public):
  if signedmessage[0] == '0':
    signempty.open(signedmessage[1],public[0:32])
    return 0
  if signedmessage[0] == '1':
    signempty.open(signedmessage[1],public[32:64])
    return 1
  raise Exception('message must be 0 or 1')
```

A signature scheme for 1-bit messages: verification

Rest of signbit.py

```
def open(signedmessage,public):
  if signedmessage[0] == '0':
    signempty.open(signedmessage[1],public[0:32])
    return 0
  if signedmessage[0] == '1':
    signempty.open(signedmessage[1],public[32:64])
    return 1
  raise Exception('message must be 0 or 1')
```

```
>>> import signbit
>>> pk,sk = signbit.keypair()
>>> sm = signbit.sign(1,sk)
>>> signbit.open(sm,pk)
1
```

A signature scheme for 4-bit messages: key generation

First part of sign4bits.py

```
import signbit

def keypair():
  p0,s0 = signbit.keypair()
  p1,s1 = signbit.keypair()
  p2,s2 = signbit.keypair()
  p3,s3 = signbit.keypair()
  return p0+p1+p2+p3,s0+s1+s2+s3
```

A signature scheme for 4-bit messages: sign & verify

Rest of sign4bits.py

```
def sign(m,secret):
  if type(m) != int: raise Exception('message must be int')
  if m < 0 or m > 15:
    raise Exception('message must be between 0 and 15')
  sm0 = signbit.sign(1 & (m >> 0),secret[0:64])
  sm1 = signbit.sign(1 & (m >> 1),secret[64:128])
  sm2 = signbit.sign(1 & (m >> 2),secret[128:192])
  sm3 = signbit.sign(1 & (m >> 3),secret[192:256])
  return sm0+sm1+sm2+sm3

def open(sm,public):
  m0 = signbit.open(sm[0:2],public[0:64])
  m1 = signbit.open(sm[2:4],public[64:128])
  m2 = signbit.open(sm[4:6],public[128:192])
  m3 = signbit.open(sm[6:],public[192:256])
  return m0 + 2*m1 + 4*m2 + 8*m3
```

Do not use one secret key to sign two messages!

```
>>> import sign4bits
>>> pk,sk = sign4bits.keypair()
>>> sm11 = sign4bits.sign(11,sk)
>>> sign4bits.open(sm11,pk)
11
>>> sm7 = sign4bits.sign(7,sk)
>>> sign4bits.open(sm7,pk)
7
>>> forgery = sm7[:6] + sm11[6:]
>>> sign4bits.open(forgery,pk)
15
```

Lamport's 1-time signature system

Sign arbitrary-length message by signing its 256-bit hash:

```
def keypair():
  keys = [signbit.keypair() for n in range(256)]
  public,secret = zip(*keys)
  return  public,secret

def sign(message,secret):
  msg = message.to_bytes(200, byteorder="little")
  h = sha3_256(msg)
  hbits = [1 & (h[i//8])>>(i%8) for i in range(256)]
  sigs = [signbit.sign(hbits[i],secret[i]) for i in range(256)]
  return sigs, message

def open(sm,public):
  message = sm[1]
  msg = message.to_bytes(200, byteorder="little")
  h = sha3_256(msg)
  hbits = [1 & (h[i//8])>>(i%8) for i in range(256)]
  for i in range(256):
    if hbits[i] != signbit.open(sm[0][i],public[i]):
      raise Exception('bit %d of hash does not match' % i)
  return message
```

Want to sign 4 bits with just 32 bytes

- Lamport's signatures have 2×256 hash outputs (each 32 bytes) as public key and the signature has 256 times 32 bytes.
- Define
$$H^i(x) = H(H^{i-1}(x)) = \underbrace{H(H(\ldots(H(x))))}_{i \text{ times}}.$$
- Pick random sk, compute pk$= H^{16}(\text{sk})$.
- For message m reveal $s = H^m(sk)$ as signature.
- To verify check that pk$= H^{16-m}(s)$.

Weak Winternitz

```
def keypair():
  secret = sha3_256(os.urandom(32))
  public = sha3_256(secret)
  for i in range(16): public = sha3_256(public)
  return public,secret

def sign(m,secret):
  if type(m) != int: raise Exception('message must be int')
  if m < 0 or m > 15: raise Exception('message must be between 0
  sign = secret
  for i in range(m): sign = sha3_256(sign)
  return sign, m

def open(sm,public):
  if type(sm[1]) != int: raise Exception('message must be int')
  if sm[1] < 0 or sm[1] > 15: raise Exception('message must be b
  check = sm[0]
  for i in range(16-sm[1]): check = sha3_256(check)
  if sha3_256(check) != public: raise Exception('bad signature')
  return sm[1]
```

Want to sign 4 bits with just 32 bytes

- Lamport's signatures have 2×256 hash outputs (each 32 bytes) as public key and the signature has 256 times 32 bytes.
- Define
$$H^i(x) = H(H^{i-1}(x)) = \underbrace{H(H(\ldots(H(x))))}_{i \text{ times}}.$$
- Pick random sk, compute pk$= H^{16}$(sk).
- For message m reveal $s = H^m(sk)$ as signature.
- To verify check that pk$= H^{16-m}(s)$.
- This works – but is insecure!

Want to sign 4 bits with just 32 bytes

- Lamport's signatures have 2×256 hash outputs (each 32 bytes) as public key and the signature has 256 times 32 bytes.
- Define
$$H^i(x) = H(H^{i-1}(x)) = \underbrace{H(H(\ldots(H(x))))}_{i \text{ times}}.$$
- Pick random sk, compute pk$= H^{16}$(sk).
- For message m reveal $s = H^m(sk)$ as signature.
- To verify check that pk$= H^{16-m}(s)$.
- This works – but is insecure!
  Eve can take $H(s)$ as signature on $m+1$ (for $m < 15$).

Want to sign 4 bits with just 32 bytes

- Lamport's signatures have 2×256 hash outputs (each 32 bytes) as public key and the signature has 256 times 32 bytes.
- Define
$$H^i(x) = H(H^{i-1}(x)) = \underbrace{H(H(\ldots(H(x))))}_{i \text{ times}}.$$
- Pick random sk, compute pk$= H^{16}(\text{sk})$.
- For message m reveal $s = H^m(sk)$ as signature.
- To verify check that pk$= H^{16-m}(s)$.
- This works – but is insecure!
Eve can take $H(s)$ as signature on $m+1$ (for $m < 15$).
- Fix by doubling the key-sizes again, running one chain forward, one in reverse.

Slow Winternitz 1-time signature system for 4 bits

Could stop at 15 iterations, but convenient to reuse code here:

```
import weak_winternitz
def keypair():
  keys = [weak_winternitz.keypair() for n in range(2)]
  public,secret = zip(*keys)
  return public,secret

def sign(m,secret):
  sign0 = weak_winternitz.sign(m,secret[0])
  sign1 = weak_winternitz.sign(16-m,secret[1])
  return sign0, sign1, m

def open(sm,public):
  m0 = weak_winternitz.open(sm[0],public[0])
  m1 = weak_winternitz.open(sm[1],public[1])
  if m0 != sm[2] or  m1 != (16-sm[2]): raise Exception('Invalid
  return sm[2]
```

Winternitz 1-time signature system

- Define parameter w. Each chain will run for 2^w steps.
- For signing a 256-bit hash this needs $t_1 = \lceil 256/w \rceil$ chains. Write m in base 2^w (integers of w bits):

$$m = (m_{t_1-1}, \ldots, m_1, m_0)$$

(zero–padding if necessary).

- Put

$$c = \sum_{i=0}^{t_1-1} (2^w - m_i)$$

Note that $c \leq t_1 2^w$.

- The checksum c gets larger if m_i is smaller.
- Write c in base 2^w. This takes $t_2 = 1 + \lceil \lfloor (\log_2 t_1 \rfloor + 1)/w \rceil$ w-bit integers

$$c = (c_{t_2-1}, \ldots, c_1, c_0).$$

- Publish $t_1 + t_2$ public keys, sign with chains of lengths

$$m_{t_1-1}, \ldots, m_1, m_0, c_{t_2-1}, \ldots, c_1, c_0.$$

Winternitz 1-time signature system for $w = 8$

- Define parameter $w = 8$. Each chain will run for $2^8 = 256$ steps.
- For signing a 256-bit hash this needs $t_1 = \lceil 256/8 \rceil = 32$ chains. Write m in base 2^8 (integers of 8 bits):

$$m = (m_{31}, \ldots, m_1, m_0)$$

(zero–padding if necessary).

- Put

$$c = \sum_{i=0}^{31} (2^8 - m_i)$$

Note that $c \leq 32 \cdot 2^8 = 2^{13}$.

- The checksum c gets larger if m_i is smaller.
- Write c in base 2^8. This takes $t_2 = 1 + \lceil (5+1)/8 \rceil = 2$ 8-bit integers

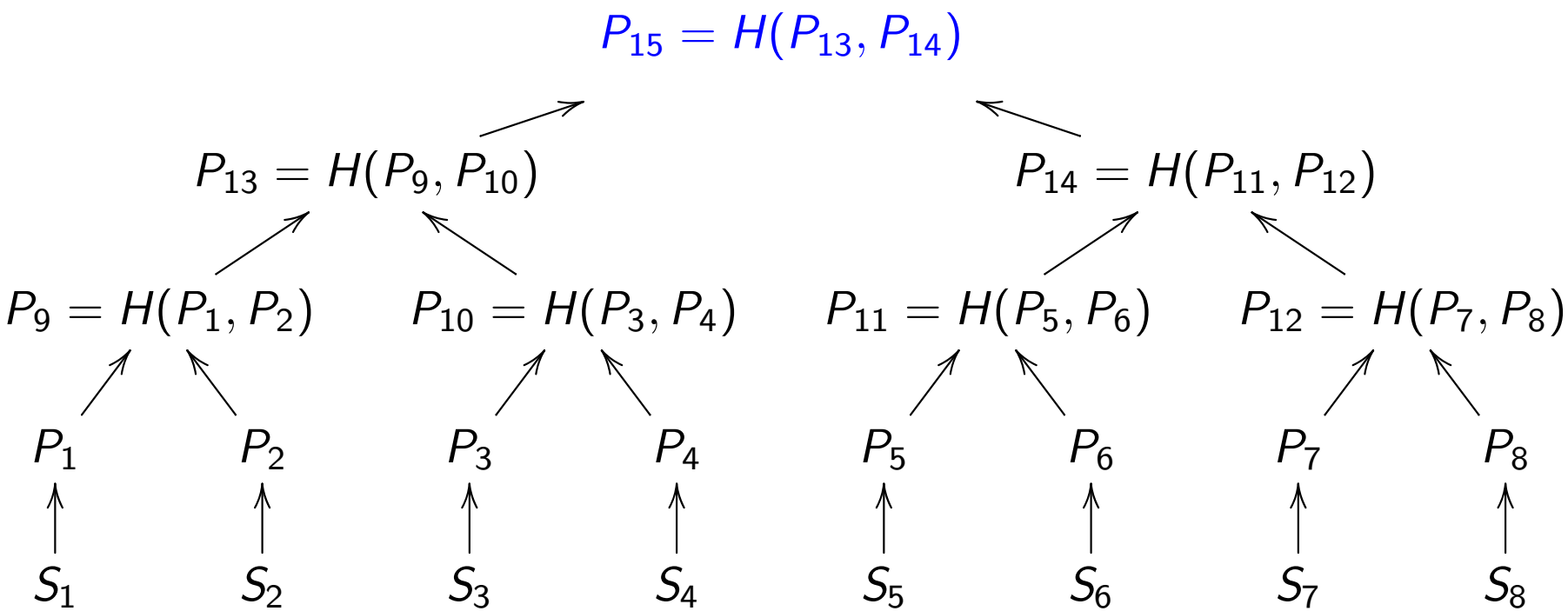
$$c = (c_1, c_0).$$

- Publish $t_1 + t_2 = 34$ public keys, sign with chains of lengths

$$m_{31}, \ldots, m_1, m_0, c_1, c_0.$$

Merkle's (e.g.) 8-time signature system

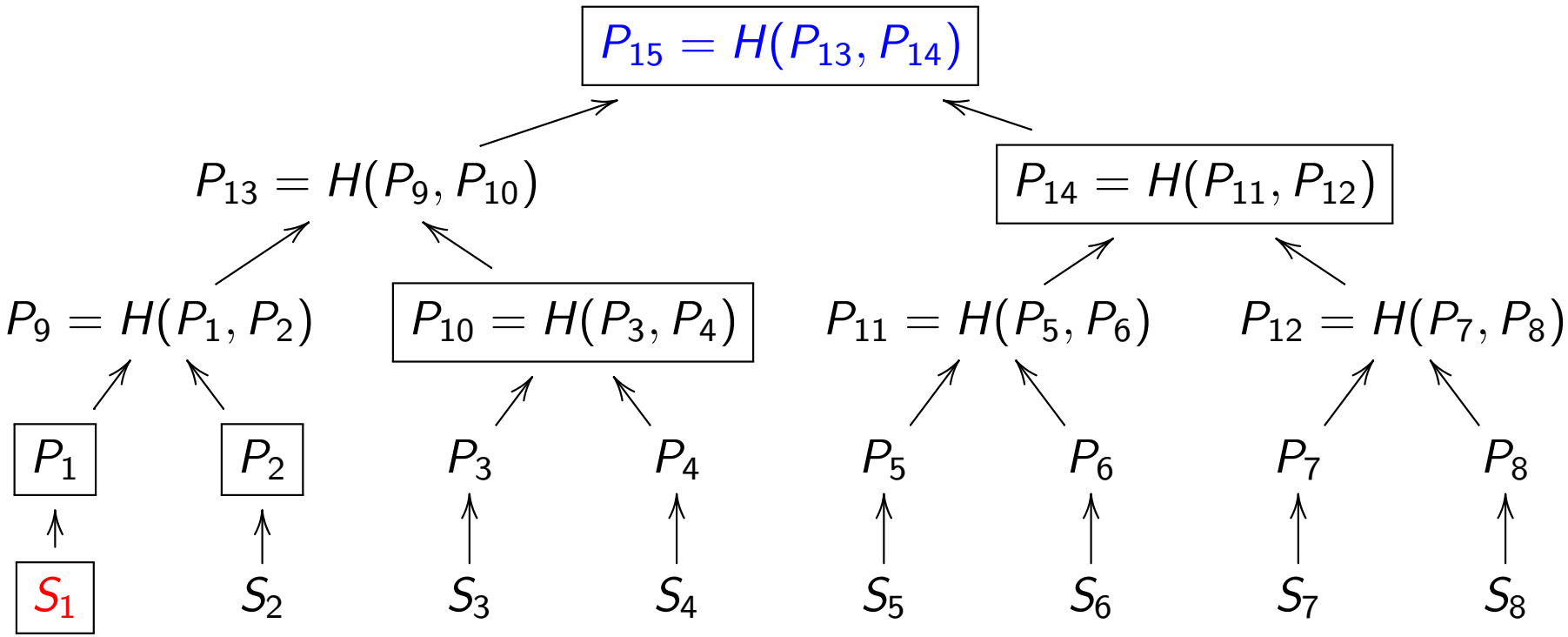
Hash 8 one-time public keys into a single Merkle public key P_{15}.



$S_i \to P_i$ can be Lamport or Winternitz one-time signature system.
Each such pair (S_i, P_i) may be used only once.

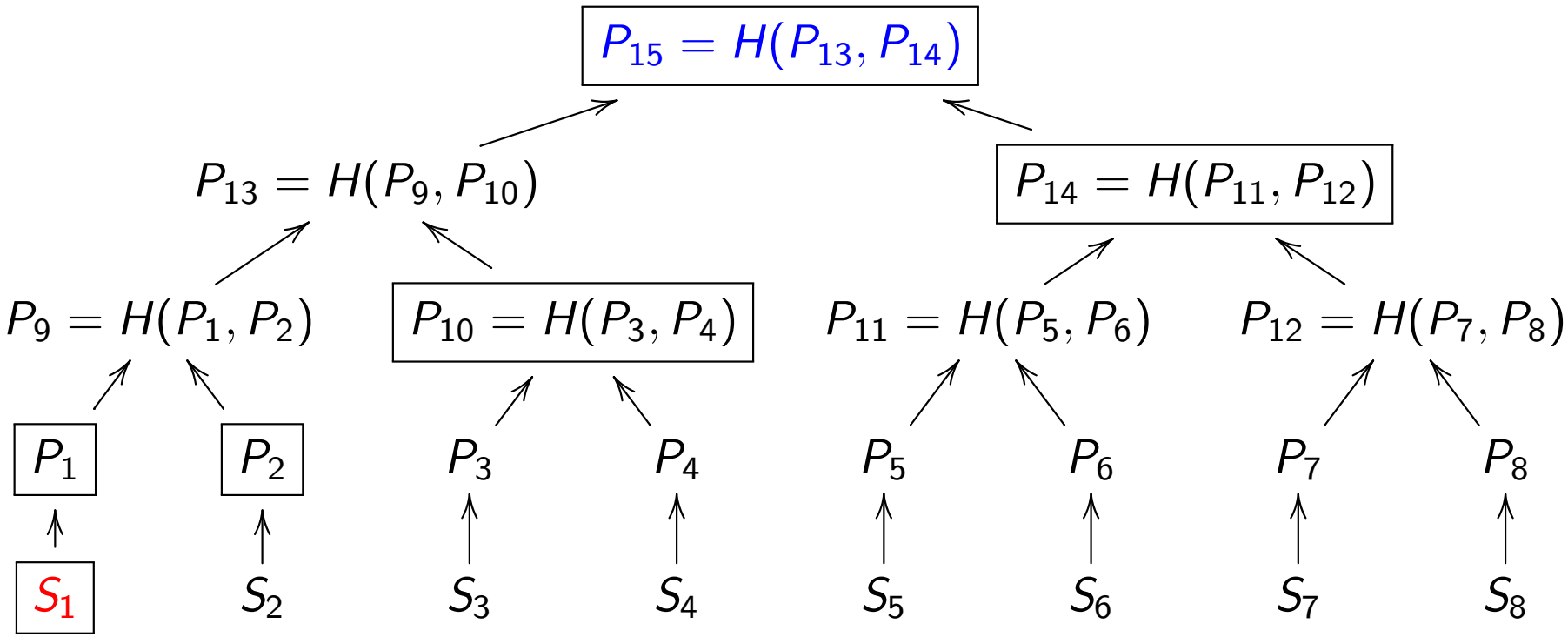
Signature in 8-time Merkle hash tree

Signature of first message: $(\texttt{sign}(m, S_1), P_1, P_2, P_{10}, P_{14})$.



Signature in 8-time Merkle hash tree

Signature of first message: $(\texttt{sign}(m, S_1), P_1, P_2, P_{10}, P_{14})$.



Verify signature $\texttt{sign}(m, S_1)$ with public key P_1 (provided in signature).
Link P_1 against public key P_{15} by computing $P'_9 = H(P_1, P_2)$,
$P'_{13} = H(P'_9, P_{10})$, and comparing $H(P'_{13}, P_{14})$ with P_{15}.
Reject if $H(P'_{13}, P_{14}) \neq P_{15}$ of if the signature verification failed.

Improvements to Merkle's scheme

- Each key is good only for fixed number of messages, typically 2^n.
- The public key is very short: just one hash output.
  But each signature contains n public keys along with the one-time signature.
- Computing the public key requires computing and storing 2^n one-time signature keys.

Improvements to Merkle's scheme

- Each key is good only for fixed number of messages, typically 2^n.
- The public key is very short: just one hash output.
  But each signature contains n public keys along with the one-time signature.
- Computing the public key requires computing and storing 2^n one-time signature keys.
- Can trade time for space by computing the secret keys S_i deterministically from a short secret seed.
  Very little storage for the seed but more time in signature generation.

Improvements to Merkle's scheme

- Each key is good only for fixed number of messages, typically 2^n.
- The public key is very short: just one hash output.
  But each signature contains n public keys along with the one-time signature.
- Computing the public key requires computing and storing 2^n one-time signature keys.
- Can trade time for space by computing the secret keys S_i deterministically from a short secret seed.
  Very little storage for the seed but more time in signature generation.
- Can build trees of trees where each leaf of the top tree signs the root of a tree below it. Only the top tree is needed in key generation.
  This increases the signature length (one one-time signature per tree) and signing time. See PhD thesis of Andreas Hülsing for an optimized schedule of what to store and when to precompute.

Stateful hash-based signatures

- Only one prerequisite: a good hash function, e.g. SHA3-512.
  Hash functions map long strings to fixed-length strings.
  Signature schemes use hash functions in handling plaintext.
- Old idea: 1979 Lamport one-time signatures.
- 1979 Merkle extends to more signatures.

Pros:

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

Cons:

- Biggish signature
  though some tradeoffs possible
- Stateful, i.e., ever reusing a subkey breaks security.
  Adam Langley "for most environments it's a huge foot-cannon."

Stateful hash-based signatures

- Only one prerequisite: a good hash function, e.g. SHA3-512.
  Hash functions map long strings to fixed-length strings.
  Signature schemes use hash functions in handling plaintext.
- Old idea: 1979 Lamport one-time signatures.
- 1979 Merkle extends to more signatures.

Pros:

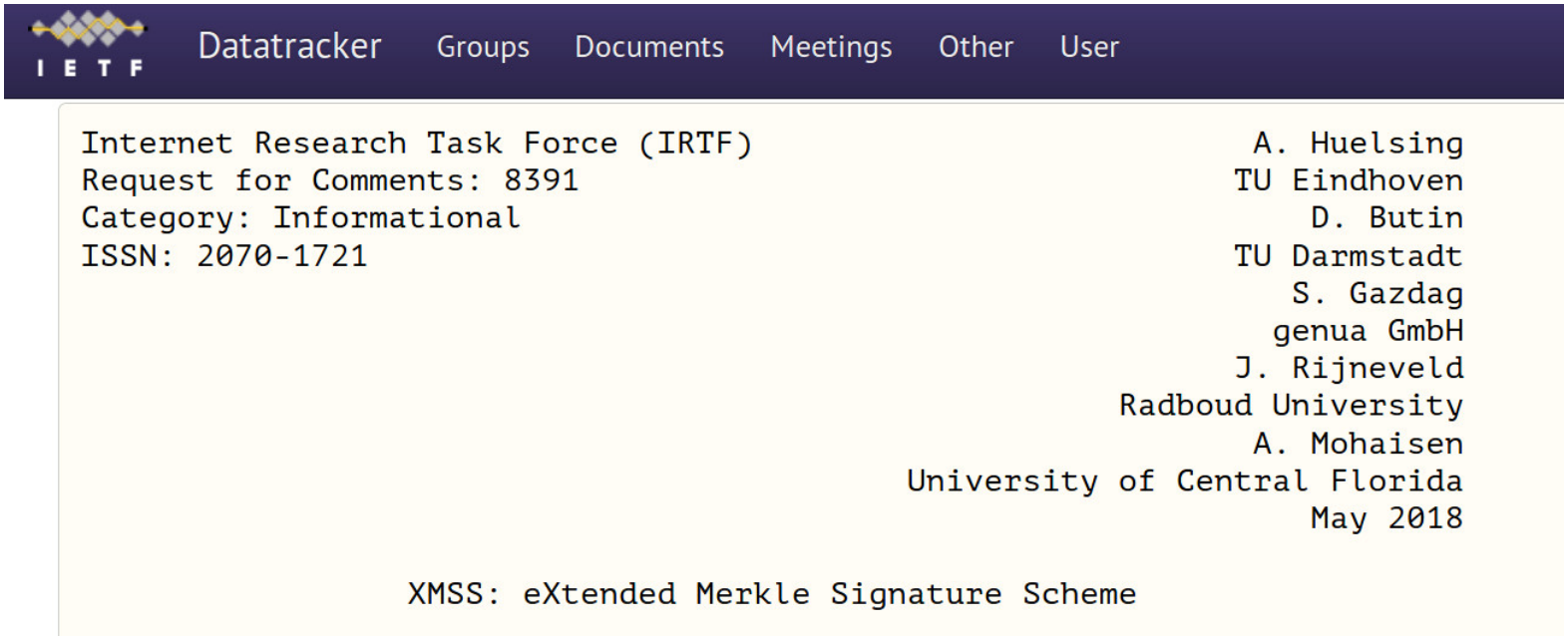
- Post quantum
- Only need secure hash function
- Security well understood
- Fast
- We can count: OS update, code signing, . . . naturally keep state.

Cons:

- Biggish signature
  though some tradeoffs possible
- Stateful, i.e., ever reusing a subkey breaks security.
  Adam Langley "for most environments it's a huge foot-cannon."

Standardization progress

- CFRG has published 2 RFCs: RFC 8391 and RFC 8554



IETF Datatracker Groups Documents Meetings Other User

Internet Research Task Force (IRTF) — A. Huelsing, TU Eindhoven; D. Butin, TU Darmstadt; S. Gazdag, genua GmbH; J. Rijneveld, Radboud University; A. Mohaisen, University of Central Florida

Request for Comments: 8391
Category: Informational
ISSN: 2070-1721

May 2018

XMSS: eXtended Merkle Signature Scheme

IETF Datatracker Groups Documents Meetings Other User

Internet Research Task Force (IRTF) — D. McGrew, M. Curcio, S. Fluhrer, Cisco Systems

Request for Comments: 8554
Category: Informational
ISSN: 2070-1721

April 2019

Leighton-Micali Hash-Based Signatures

Standardization progress

- CFRG has published 2 RFCs: RFC 8391 and RFC 8554
- NIST has gone through two rounds of requests for public input, most are positive and recommend standardizing XMSS and LMS. Only concern is about statefulness in general.

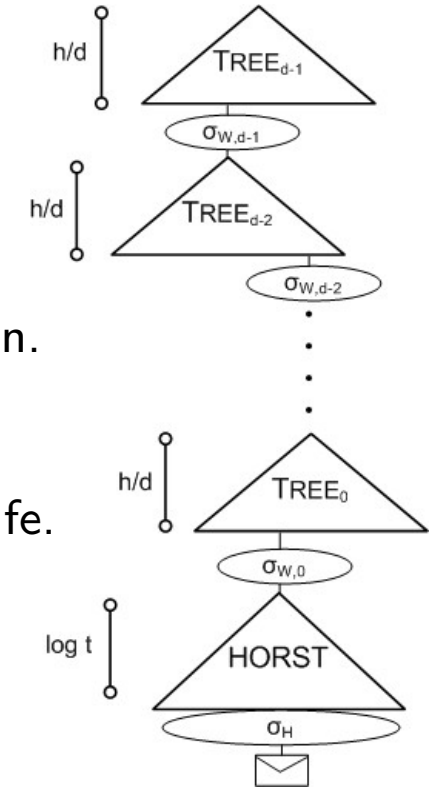
Standardization progress

- CFRG has published 2 RFCs: RFC 8391 and RFC 8554
- NIST has gone through two rounds of requests for public input, most are positive and recommend standardizing XMSS and LMS. Only concern is about statefulness in general.

- ISO SC27 JTC1 WG2 has started a study period on stateful hash-based signatures.

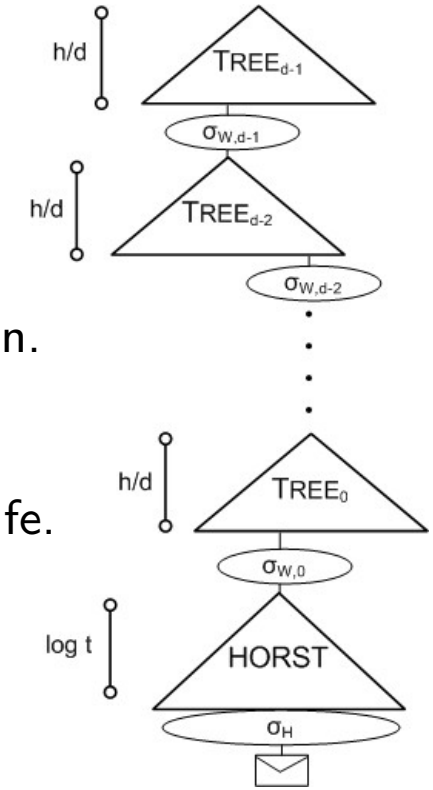
Stateless hash-based signatures

- Idea from 1987 Goldreich:
  - Signer builds huge tree of certificate authorities.
  - Signature includes certificate chain.
  - Each CA is a hash of master secret and tree position.
    This is deterministic, so don't need to store results.
  - **Random** bottom-level CA signs message.
    Many bottom-level CAs, so one-time signature is safe.

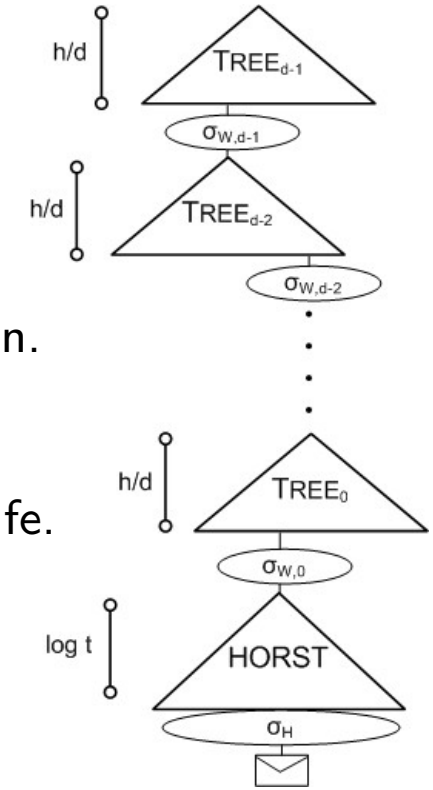


Stateless hash-based signatures

- Idea from 1987 Goldreich:
  - Signer builds huge tree of certificate authorities.
  - Signature includes certificate chain.
  - Each CA is a hash of master secret and tree position.
    This is deterministic, so don't need to store results.
  - **Random** bottom-level CA signs message.
    Many bottom-level CAs, so one-time signature is safe.
- 0.6 MB: Goldreich's signature with good 1-time signature scheme.
- 1.2 MB: average Debian package size.
- 1.8 MB: average web page in Alexa Top 1000000.



Stateless hash-based signatures



- Idea from 1987 Goldreich:
  - Signer builds huge tree of certificate authorities.
  - Signature includes certificate chain.
  - Each CA is a hash of master secret and tree position.
    This is deterministic, so don't need to store results.
  - **Random** bottom-level CA signs message.
    Many bottom-level CAs, so one-time signature is safe.
- 0.6 MB: Goldreich's signature with good 1-time signature scheme.
- 1.2 MB: average Debian package size.
- 1.8 MB: average web page in Alexa Top 1000000.
- 0.041 MB: SPHINCS signature, new optimization of Goldreich.
  Modular, guaranteed as strong as its components (hash, PRNG).
  Well-known components chosen for 2^{128} post-quantum security.
  `sphincs.cr.yp.to`

NIST submission SPHINCS+

- Signature based on hash functions.
- Requires only a secure hash function, no further assumptions.
- Based on ideas of Lamport (1979) and Merkle (1979).
- Developed starting from SPHINCS with
  - improve multi-signature,
  - smaller keys,
  - Option for shorter signatures (30kB instead of 41kB) if "only" 2^{50} messages signed.
- Three versions (using different hash functions)
  - SPHINCS+-SHA3 (with SHAKE256),
  - SPHINCS+-SHA2 (with SHA-256),
  - SPHINCS+-Haraka (with Haraka, a hash function for short inputs).

More info at https://sphincs.org/.

Initial recommendations of long-term secure post-quantum systems

Daniel Augot, Lejla Batina, Daniel J. Bernstein, Joppe Bos, Johannes Buchmann, Wouter Castryck, Orr Dunkelman, Tim Güneysu, Shay Gueron, Andreas Hülsing, Tanja Lange, Mohamed Saied Emam Mohamed, Christian Rechberger, Peter Schwabe, Nicolas Sendrier, Frederik Vercauteren, Bo-Yin Yang

Initial recommendations

- **Symmetric encryption** Thoroughly analyzed, 256-bit keys:
  - AES-256
  - Salsa20 with a 256-bit key

  Evaluating: Serpent-256, ...
- **Symmetric authentication** Information-theoretic MACs:
  - GCM using a 96-bit nonce and a 128-bit authenticator
  - Poly1305
- **Public-key encryption** McEliece with binary Goppa codes:
  - length $n = 6960$, dimension $k = 5413$, $t = 119$ errors

  Evaluating: QC-MDPC, Stehlé-Steinfeld NTRU, ...
- **Public-key signatures** Hash-based (minimal assumptions):
  - XMSS with any of the parameters specified in CFRG draft
  - SPHINCS-256

  Evaluating: HFEv-, ...