Post-quantum cryptography

Tanja Lange

Technische Universiteit Eindhoven

15 December 2022

with some slides from joint talks with Daniel J. Bernstein

Algorithms for Quantum Computation: Discrete Logarithms and Factoring

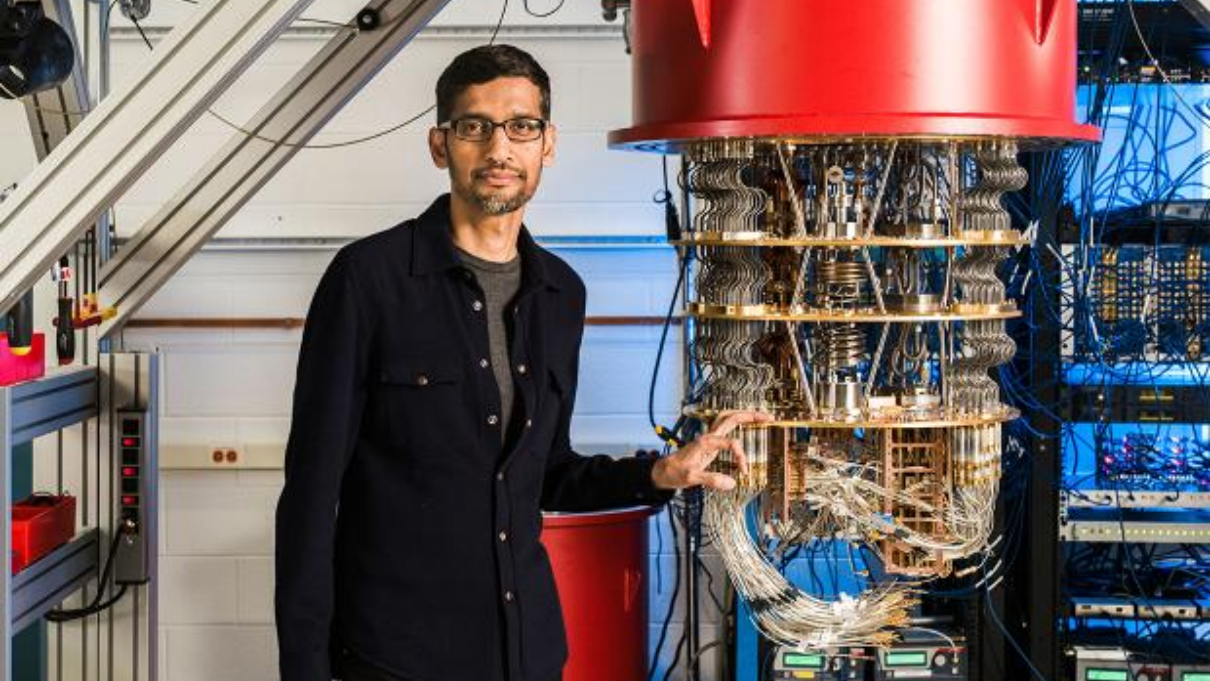
Peter W. Shor
AT&T Bell Labs
Room 2D-149
600 Mountain Ave.
Murray Hill, NJ 07974, USA

Abstract

A computer is generally considered to be a universal computational device; i.e., it is believed able to simulate any physical computational device with a cost in computation time of at most a polynomial factor. It is not clear whether this is still true when quantum mechanics is taken into consideration. Several researchers, starting with David Deutsch, have developed models for quantum mechanical computers and have investigated their compu-

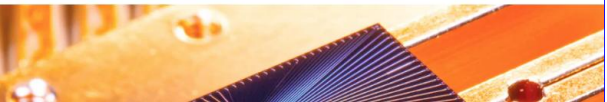
[1, 2]. Although he did not ask whether quantum mechanics conferred extra power to computation, he did show that a Turing machine could be simulated by the reversible unitary evolution of a quantum process, which is a necessary prerequisite for quantum computation. Deutsch [9, 10] was the first to give an explicit model of quantum computation. He defined both quantum Turing machines and quantum circuits and investigated some of their properties.

The next part of this paper discusses how quantum com-



Premium

Quantum computing could end encryption within five years, says Google boss



Mr Pichai said a combination of artificial intelligence and quantum would "help us tackle some of the biggest problems we see", but said it was important encryption evolved to match this.

"In a five to ten year time frame, quantum computing will break encryption as we know it today."

This is because current encryption methods, by which information such as texts or passwords is turned into code to make it unreadable, rely upon the fact that classic computers would take billions of years to decipher that code.

Quantum computers, with their ability to be

Commonly used systems

Sender
"Alice"

Untrustworthy network
"Eve"

Receiver
"Bob"

Cryptography with symmetric keys
AES-128. **AES-192**. **AES-256**. **AES-GCM**. **ChaCha20**. **HMAC-SHA-256**. **Poly1305**. **SHA-2**. **SHA-3**. **Salsa20**.

Cryptography with public keys
BN-254. **Curve25519**. **DH**. **DSA**. **ECDH**. **ECDSA**. **EdDSA**. **NIST P-256**. **NIST P-384**. **NIST P-521**. **RSA encrypt**. **RSA sign**. **secp256k1**.

Commonly used systems

Sender
"Alice"

Untrustworthy network
"Eve" with quantum computer

Receiver
"Bob"

Cryptography with symmetric keys
AES-128. **AES-192**. **AES-256**. **AES-GCM**. **ChaCha20**. **HMAC-SHA-256**. **Poly1305**. **SHA-2**. **SHA-3**. **Salsa20**.

Cryptography with public keys
BN-254. **Curve25519**. **DH**. **DSA**. **ECDH**. **ECDSA**. **EdDSA**. **NIST P-256**. **NIST P-384**. **NIST P-521**. **RSA encrypt**. **RSA sign**. **secp256k1**.

Symmetric-key authenticated encryption

Sender
"Alice"

Untrustworthy network
"Eve" with quantum computer

Receiver
"Bob"

- Very easy solutions **if Alice and Bob already share long secret key k**:
  - "One-time pad" for confidentiality.
  - "Wegman–Carter MAC" for integrity and authenticity.
- AES-256: Standardized method to expand **short secret key** (256-bit k) into string indistinguishable from long secret key.
- AES introduced in 1998 by Daemen and Rijmen.
  Security analyzed in papers by dozens of cryptanalysts.
- No credible threat from quantum algorithms. Grover costs 2^{128}.
- Some results assume attacker has quantum access to computation, then some systems are weaker . . . but I'd know if my laptop had turned into a quantum computer.

National Academy of Sciences (US)

4 December 2018: Report on quantum computing

Don't panic. "Key Finding 1: Given the current state of quantum computing and recent rates of progress, it is highly unexpected that a quantum computer that can compromise RSA 2048 or comparable discrete logarithm-based public key cryptosystems will be built within the next decade."

National Academy of Sciences (US)

4 December 2018: Report on quantum computing

Don't panic. "Key Finding 1: Given the current state of quantum computing and recent rates of progress, it is highly unexpected that a quantum computer that can compromise RSA 2048 or comparable discrete logarithm-based public key cryptosystems will be built within the next decade."

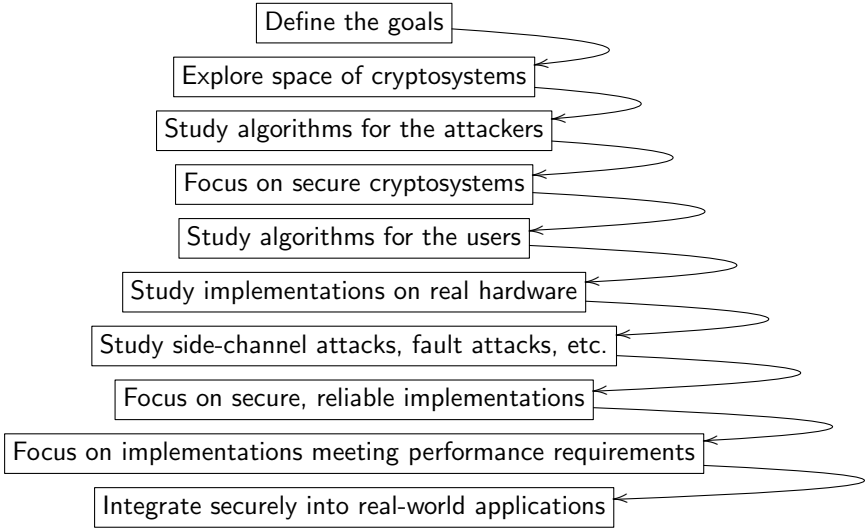
Panic. "Key Finding 10: Even if a quantum computer that can decrypt current cryptographic ciphers is more than a decade off, the hazard of such a machine is high enough—and the time frame for transitioning to a new security protocol is sufficiently long and uncertain—that prioritization of the development, standardization, and deployment of post-quantum cryptography is critical for minimizing the chance of a potential security and privacy disaster."

"[Section 4.4:] In particular, all encrypted data that is recorded today and stored for future use, will be cracked once a large-scale quantum computer is developed."

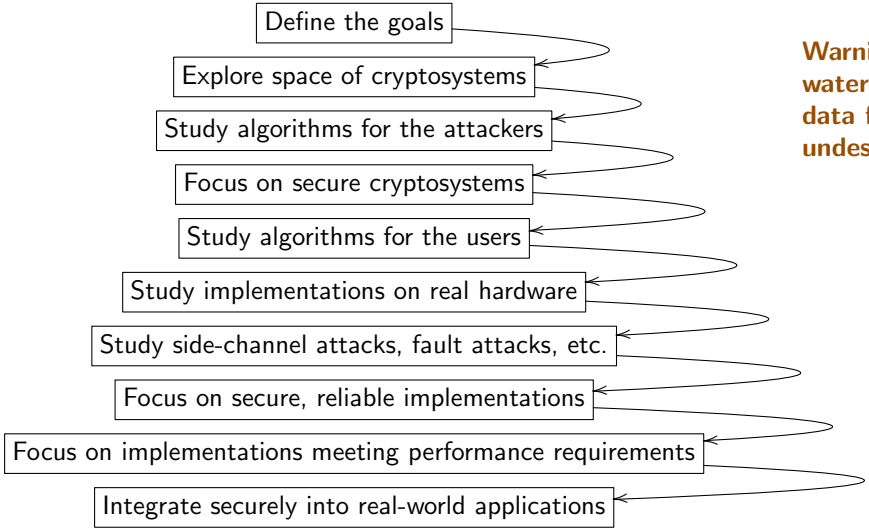
High urgency for long-term confidentiality

- Today's encrypted communication is being stored by attackers and will be decrypted years later with quantum computers. Danger for human-rights workers, medical records, journalists, security research, legal proceedings, state secrets, ...

Many stages of research from design to deployment



Many stages of research from design to deployment



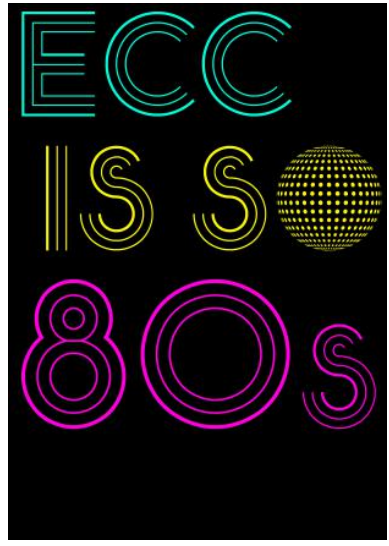
Warning: waterfall data flow, undesirable.

Confidence-inspiring crypto takes time to build

- Example: ECC introduced **1985**; big advantages over RSA. Robust ECC started to take over the Internet in **2015**.
- Can't wait for quantum computers before finding a solution!

Confidence-inspiring crypto takes time to build

- Example: ECC introduced **1985**; big advantages over RSA. Robust ECC started to take over the Internet in **2015**.
- Can't wait for quantum computers before finding a solution!



Post-quantum cryptography

Cryptography under the assumption that the attacker has a quantum computer.

Post-quantum cryptography

Cryptography under the assumption that the attacker has a quantum computer.

- 1994: Shor's quantum algorithm. 1996: Grover's quantum algorithm.
  Many subsequent papers on quantum algorithms: see quantumalgorithmzoo.org.
- 2003: Daniel J. Bernstein introduces term Post-quantum cryptography.
- 2006: First International Workshop on Post-Quantum Cryptography. PQCrypto 2006, 2008, 2010, 2011, 2013, 2014, 2016, 2017, 2018, 2019, 2020, 2021, (soon) 2022.
- 2015: NIST hosts its first workshop on post-quantum cryptography.
- 2016: NIST announces a standardization project for post-quantum systems.
- 2017: Deadline for submissions to the NIST competition.
- 2019: Second round of NIST competition begins.
- 2020: Third round of NIST competition begins.
- 2021

Post-quantum cryptography

Cryptography under the assumption that the attacker has a quantum computer.

- 1994: Shor's quantum algorithm. 1996: Grover's quantum algorithm.
  Many subsequent papers on quantum algorithms: see quantumalgorithmzoo.org.
- 2003: Daniel J. Bernstein introduces term Post-quantum cryptography.
- 2006: First International Workshop on Post-Quantum Cryptography. PQCrypto 2006, 2008, 2010, 2011, 2013, 2014, 2016, 2017, 2018, 2019, 2020, 2021, (soon) 2022.
- 2015: NIST hosts its first workshop on post-quantum cryptography.
- 2016: NIST announces a standardization project for post-quantum systems.
- 2017: Deadline for submissions to the NIST competition.
- 2019: Second round of NIST competition begins.
- 2020: Third round of NIST competition begins.
- ~~2021~~ 2022 "not later than the end of March'

Post-quantum cryptography

Cryptography under the assumption that the attacker has a quantum computer.

- 1994: Shor's quantum algorithm. 1996: Grover's quantum algorithm.
  Many subsequent papers on quantum algorithms: see quantumalgorithmzoo.org.
- 2003: Daniel J. Bernstein introduces term Post-quantum cryptography.
- 2006: First International Workshop on Post-Quantum Cryptography. PQCrypto 2006, 2008, 2010, 2011, 2013, 2014, 2016, 2017, 2018, 2019, 2020, 2021, (soon) 2022.
- 2015: NIST hosts its first workshop on post-quantum cryptography.
- 2016: NIST announces a standardization project for post-quantum systems.
- 2017: Deadline for submissions to the NIST competition.
- 2019: Second round of NIST competition begins.
- 2020: Third round of NIST competition begins.
- ~~2021~~ 2022 ~~"not later than the end of March"~~:

Post-quantum cryptography

Cryptography under the assumption that the attacker has a quantum computer.

- 1994: Shor's quantum algorithm. 1996: Grover's quantum algorithm.
  Many subsequent papers on quantum algorithms: see quantumalgorithmzoo.org.
- 2003: Daniel J. Bernstein introduces term Post-quantum cryptography.
- 2006: First International Workshop on Post-Quantum Cryptography. PQCrypto 2006, 2008, 2010, 2011, 2013, 2014, 2016, 2017, 2018, 2019, 2020, 2021, (soon) 2022.
- 2015: NIST hosts its first workshop on post-quantum cryptography.
- 2016: NIST announces a standardization project for post-quantum systems.
- 2017: Deadline for submissions to the NIST competition.
- 2019: Second round of NIST competition begins.
- 2020: Third round of NIST competition begins.
- ~~2021~~ 2022 ~~"not later than the end of March"~~: 05 Jul NIST announces first selections.
- 2022 $\rightarrow \infty$ NIST studies further systems.
- 2023/2024?: NIST issues post-quantum standards.

Major categories of public-key post-quantum systems

- **Code-based** encryption: McEliece cryptosystem has survived since 1978. Short ciphertexts and large public keys. Security relies on hardness of decoding error-correcting codes.
- **Hash-based** signatures: very solid security and small public keys. Require only a secure hash function (hard to find second preimages).
- **Isogeny-based** encryption: new kid on the block, promising short keys and ciphertexts and non-interactive key exchange. Security relies on hardness of finding isogenies between elliptic curves over finite fields.
- **Lattice-based** encryption and signatures: possibility for balanced sizes. Security relies on hardness of finding short vectors in some (typically special) lattice.
- **Multivariate-quadratic** signatures: short signatures and large public keys. Security relies on hardness of solving systems of multivariate equations over finite fields.

Warning: These are categories of mathematical problems;
individual systems may be totally insecure if the problem is not used correctly.

We have a good algorithmic abstraction of what a quantum computer can do,
but new systems need more analysis. Any extra structure offers more attack surface.

NIST's 5 July announcement

The winners:

- Kyber, a KEM based on structured lattices
- Dilithium, a signature scheme based on structured lattices
- Falcon, a signature scheme based on structured lattices
- SPHINCS+, a signature scheme based on

NIST's 5 July announcement

The winners:

- Kyber, a KEM based on structured lattices
- Dilithium, a signature scheme based on structured lattices
- Falcon, a signature scheme based on structured lattices
- SPHINCS+, a signature scheme based on hash functions

NIST's 5 July announcement

The winners:

- Kyber, a KEM based on structured lattices
- Dilithium, a signature scheme based on structured lattices
- Falcon, a signature scheme based on structured lattices
- SPHINCS+, a signature scheme based on hash functions

This is an odd choice, given that KEMs are most urgently needed to ensure long-term confidentiality.

NIST's 5 July announcement

The winners:

- Kyber, a KEM based on structured lattices
- Dilithium, a signature scheme based on structured lattices
- Falcon, a signature scheme based on structured lattices
- SPHINCS+, a signature scheme based on hash functions

This is an odd choice, given that KEMs are most urgently needed to ensure long-term confidentiality.

Schemes advancing to round 4, so maybe more winners later:

- BIKE, a KEM based on codes
- Classic McEliece, a KEM based on codes
- HQC, a KEM based on codes
- SIKE, a KEM based on isogenies (now really badly broken, < 1 month after NIST's announcement)

21 December 2017: NIST posts 69 submissions from 260 people.

BIG QUAKE. **BIKE**. **CFPKM**. **Classic McEliece**. **Compact LWE**. **CRYSTALS-DILITHIUM**. **CRYSTALS-KYBER**. **DAGS**. **Ding Key Exchange**. **DME**. **DRS**. **DualModeMS**. **Edon-K**. **EMBLEM and R.EMBLEM**. **FALCON**. **FrodoKEM**. **GeMSS**. **Giophantus**. **Gravity-SPHINCS**. **Guess Again**. **Gui**. **HILA5**. **HiMQ-3**. **HK17**. **HQC**. **KINDI**. **LAC**. **LAKE**. **LEDAkem**. **LEDApkc**. **Lepton**. **LIMA**. **Lizard**. **LOCKER**. **LOTUS**. **LUOV**. **McNie**. **Mersenne-756839**. **MQDSS**. **NewHope**. **NTRU Prime**. **NTRU-HRSS-KEM**. **NTRUEncrypt**. **NTS-KEM**. **Odd Manhattan**. **OKCN/AKCN/CNKE**. **Ouroboros-R**. **Picnic**. **pqNTRUSign**. **pqRSA encryption**. **pqRSA signature**. **pqsigRM**. **QC-MDPC KEM**. **qTESLA**. **RaCoSS**. **Rainbow**. **Ramstake**. **RankSign**. **RLCE-KEM**. **Round2**. **RQC**. **RVB**. **SABER**. **SIKE**. **SPHINCS+**. **SRTPI**. **Three Bears**. **Titanium**. **WalnutDSA**.

By end of 2017: 8 out of 69 submissions attacked.

BIG QUAKE. **BIKE**. **CFPKM**. **Classic McEliece**. **Compact LWE**. **CRYSTALS-DILITHIUM**. **CRYSTALS-KYBER**. **DAGS**. **Ding Key Exchange**. **DME**. **DRS**. **DualModeMS**. **Edon-K**. **EMBLEM and R.EMBLEM**. **FALCON**. **FrodoKEM**. **GeMSS**. **Giophantus**. **Gravity-SPHINCS**. **Guess Again**. **Gui**. **HILA5**. **HiMQ-3**. **HK17**. **HQC**. **KINDI**. **LAC**. **LAKE**. **LEDAkem**. **LEDApkc**. **Lepton**. **LIMA**. **Lizard**. **LOCKER**. **LOTUS**. **LUOV**. **McNie**. **Mersenne-756839**. **MQDSS**. **NewHope**. **NTRU Prime**. **NTRU-HRSS-KEM**. **NTRUEncrypt**. **NTS-KEM**. **Odd Manhattan**. **OKCN/AKCN/CNKE**. **Ouroboros-R**. **Picnic**. **pqNTRUSign**. **pqRSA encryption**. **pqRSA signature**. **pqsigRM**. **QC-MDPC KEM**. **qTESLA**. **RaCoSS**. **Rainbow**. **Ramstake**. **RankSign**. **RLCE-KEM**. **Round2**. **RQC**. **RVB**. **SABER**. **SIKE**. **SPHINCS+**. **SRTPI**. **Three Bears**. **Titanium**. **WalnutDSA**.

Some less security than claimed; some really broken; some attack scripts.

By end of 2018: 22 out of 69 submissions attacked.

BIG QUAKE. **BIKE**. **CFPKM**. **Classic McEliece**. **Compact LWE**. **CRYSTALS-DILITHIUM**. **CRYSTALS-KYBER**. **DAGS**. **Ding Key Exchange**. **DME**. **DRS**. **DualModeMS**. **Edon-K**. **EMBLEM and R.EMBLEM**. **FALCON**. **FrodoKEM**. **GeMSS**. **Giophantus**. **Gravity-SPHINCS**. **Guess Again**. **Gui**. **HILA5**. **HiMQ-3**. **HK17**. **HQC**. **KINDI**. **LAC**. **LAKE**. **LEDAkem**. **LEDApkc**. **Lepton**. **LIMA**. **Lizard**. **LOCKER**. **LOTUS**. **LUOV**. **McNie**. **Mersenne-756839**. **MQDSS**. **NewHope**. **NTRU Prime**. **NTRU-HRSS-KEM**. **NTRUEncrypt**. **NTS-KEM**. **Odd Manhattan**. **OKCN/AKCN/CNKE**. **Ouroboros-R**. **Picnic**. **pqNTRUSign**. **pqRSA encryption**. **pqRSA signature**. **pqsigRM**. **QC-MDPC KEM**. **qTESLA**. **RaCoSS**. **Rainbow**. **Ramstake**. **RankSign**. **RLCE-KEM**. **Round2**. **RQC**. **RVB**. **SABER**. **SIKE**. **SPHINCS+**. **SRTPI**. **Three Bears**. **Titanium**. **WalnutDSA**.

Some less security than claimed; some really broken; some attack scripts.

30 January 2019: 26 candidates retained for second round.

. **BIKE**. . **Classic McEliece**. .
CRYSTALS-DILITHIUM. **CRYSTALS-KYBER**. . . .
. . . . **FALCON**. **FrodoKEM**.
GeMSS. . . . . **HILA5**. . .
HQC. . **LAC**. **LAKE**. **LEDAkem**. **LEDApkc**. . . . **LOCKER**.
. **LUOV**. . . **MQDSS**. **NewHope**. **NTRU Prime**.
NTRU-HRSS-KEM. **NTRUEncrypt**. **NTS-KEM**. .
. **Ouroboros-R**. **Picnic**. . .
. . . **qTESLA**. . **Rainbow**.
. . . **Round2**. **RQC**. . **SABER**. **SIKE**. **SPHINCS+**.
. **Three Bears**. . .

Some less security than claimed; some really broken; some attack scripts.
Merges for second round: HILA5 & Round2; LAKE, LOCKER, & Ouroboros-R;
LEDAkem & LEDApkc; NTRUEncrypt & NTRU-HRSS-KEM.

By end of 2019: 30 out of 69 submissions attacked.

. **BIKE**. . **Classic McEliece**. .
CRYSTALS-DILITHIUM. **CRYSTALS-KYBER**. . . .
. . . . **FALCON**. **FrodoKEM**.
GeMSS. . . . . **HILA5**. . .
HQC. . **LAC**. **LAKE**. **LEDAkem**. **LEDApkc**. . . . **LOCKER**.
. **LUOV**. . . **MQDSS**. **NewHope**. **NTRU Prime**.
NTRU-HRSS-KEM. **NTRUEncrypt**. **NTS-KEM**. .
. **Ouroboros-R**. **Picnic**. . .
. . . **qTESLA**. . **Rainbow**.
. . . **Round2**. **RQC**. . **SABER**. **SIKE**. **SPHINCS+**.
. **Three Bears**. . .

Some less security than claimed; some really broken; some attack scripts.
Merges for second round: HILA5 & Round2; LAKE, LOCKER, & Ouroboros-R;
LEDAkem & LEDApkc; NTRUEncrypt & NTRU-HRSS-KEM.

22 July 2020: 15 candidates retained for third round.

BIKE. **Classic McEliece**.
CRYSTALS-DILITHIUM. **CRYSTALS-KYBER**.
FALCON. **FrodoKEM**.
GeMSS.
HQC.
NTRU Prime.
NTRU-HRSS-KEM. **NTRUEncrypt**. **NTS-KEM**.
Picnic.
Rainbow.
SABER. **SIKE**. **SPHINCS+**.

Some less security than claimed; some really broken; some attack scripts.
Merges for second round: HILA5 & Round2; LAKE, LOCKER, & Ouroboros-R;
LEDAkem & LEDApkc; NTRUEncrypt & NTRU-HRSS-KEM.
Merges for third round: Classic McEliece & NTS-KEM.

Spring 2022: 2 out of 15 candidates attacked.

BIKE. **Classic McEliece**.
CRYSTALS-DILITHIUM. **CRYSTALS-KYBER**.
FALCON. **FrodoKEM**.
GeMSS.
HQC.
NTRU Prime.
NTRU-HRSS-KEM. **NTRUEncrypt**. **NTS-KEM**.
Picnic.
Rainbow.
SABER. **SIKE**. **SPHINCS+**.

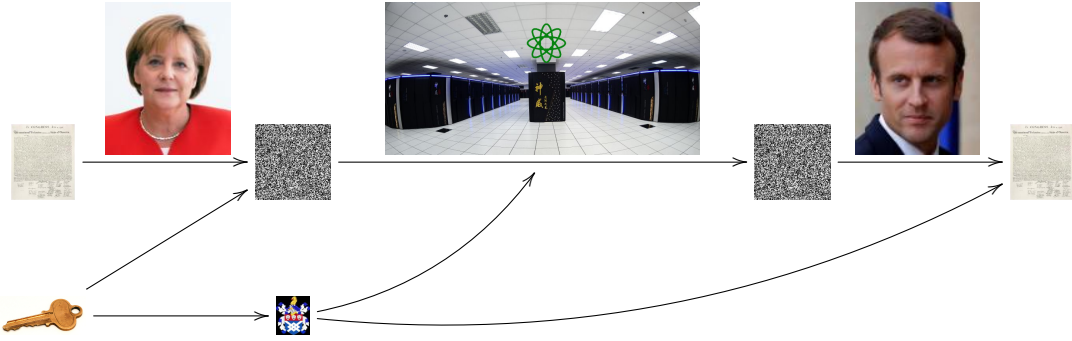
Some less security than claimed; some really broken; some attack scripts.
Merges for second round: HILA5 & Round2; LAKE, LOCKER, & Ouroboros-R;
LEDAkem & LEDApkc; NTRUEncrypt & NTRU-HRSS-KEM.
Merges for third round: Classic McEliece & NTS-KEM.
Life remains interesting in third round . . .

30 July 2020: 4 winners, 4 more candidates of which 1 broken.

BIKE. **Classic McEliece**.
CRYSTALS-DILITHIUM. **CRYSTALS-KYBER**.
FALCON.
HQC.
NTS-KEM.
SIKE. **SPHINCS+**.

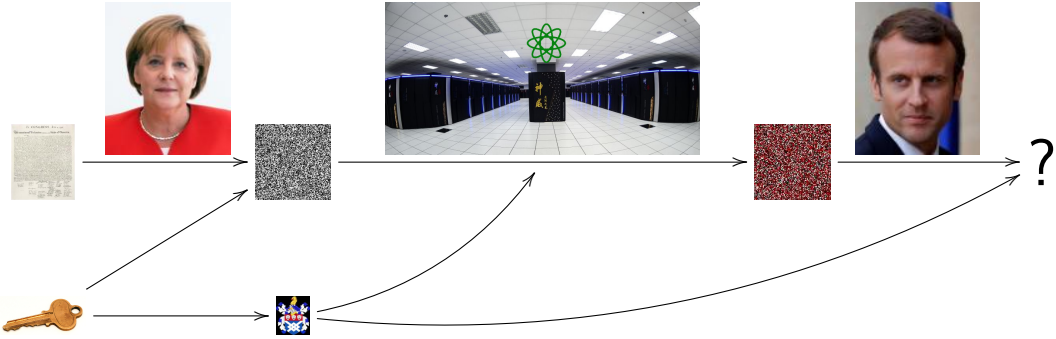
Some less security than claimed; some really broken; some attack scripts.
Merges for second round: HILA5 & Round2; LAKE, LOCKER, & Ouroboros-R;
LEDAkem & LEDApkc; NTRUEncrypt & NTRU-HRSS-KEM.
Merges for third round: Classic McEliece & NTS-KEM.
Life remains interesting in third round . . .
4 winners, 4 schemes in fourth round. SIKE completely broken with attack running in seconds.

Post-quantum public-key signatures



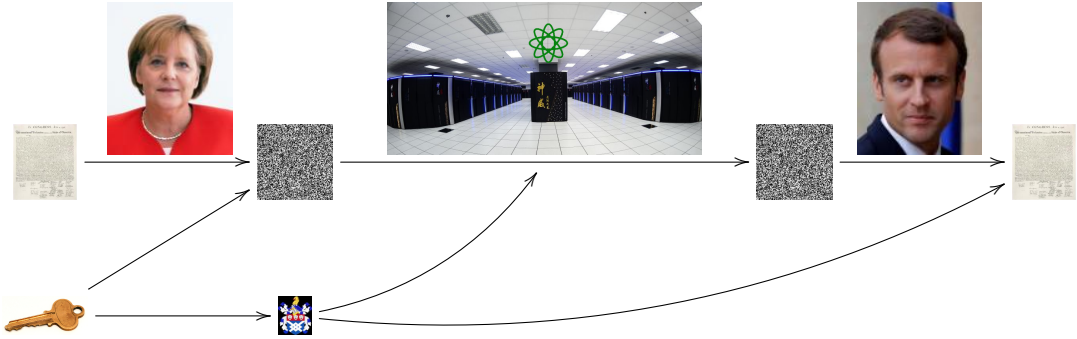
- Secret key  , public key  .

Post-quantum public-key signatures



- Secret key  , public key  .

Post-quantum public-key signatures: hash-based

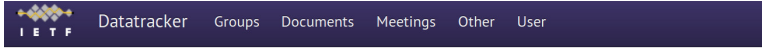


- Secret key  , public key  .
- Only one prerequisite: a good hash function, e.g. SHA3-512, ...
  Hash functions map long strings to fixed-length strings. $H : \{0,1\}^* \to \{0,1\}^n$.

  Signature schemes use hash functions in handling  .
- Quantum computers affect the hardness only marginally (Grover, not Shor).
- Old idea: 1979 Lamport one-time signatures; 1979 Merkle extends to more signatures.

On the fast track: stateful hash-based signatures

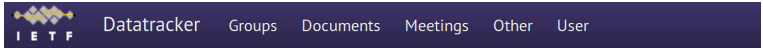
- CFRG has published 2 RFCs: RFC 8391 and RFC 8554



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



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

On the fast track: stateful hash-based signatures

- CFRG has published 2 RFCs: RFC 8391 and RFC 8554
- NIST followed both RFCs and standardized XMSS and LMS.
  Only concern is about statefulness in general.

On the fast track: stateful hash-based signatures

- CFRG has published 2 RFCs: RFC 8391 and RFC 8554
- NIST followed both RFCs and standardized XMSS and LMS.
  Only concern is about statefulness in general.

- ISO SC27 JTC1 WG2 is gettting closer to standard on stateful hash-based signatures
  (again both XMSS and LMS).

One-time signatures

A signature scheme for empty messages: key generation

A signature scheme for empty messages: key generation

First part of signempty.py

```
import os; from hashlib import sha3_256;

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; from hashlib import sha3_256;

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'a447bc8d7c661f85defcf1bbf8bad77bfc6191068a8b658c99c7ef4cbe37cf9f'
>>> binascii.hexlify(sk)
b'a4a1334a6926d04c4aa7cd98231f4b644be90303e4090c358f2946f1c257687a'
```

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
```

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 and 15')
  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 between 0 and 15')
  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$= H^{16}(\text{sk})$.
- For message m reveal $s = H^m(sk)$ as signature.
- To verify check that pk$= H^{16-m}(s)$.
- This works – but is insecure!

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 $m < 15$).

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 signature')
  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

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

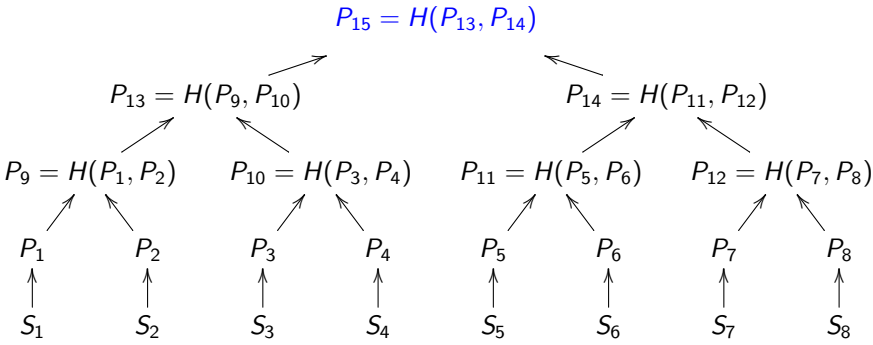
- Publish $t_1 + t_2 = 34$ public keys, sign with chains of lengths

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

More than one signature per key

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

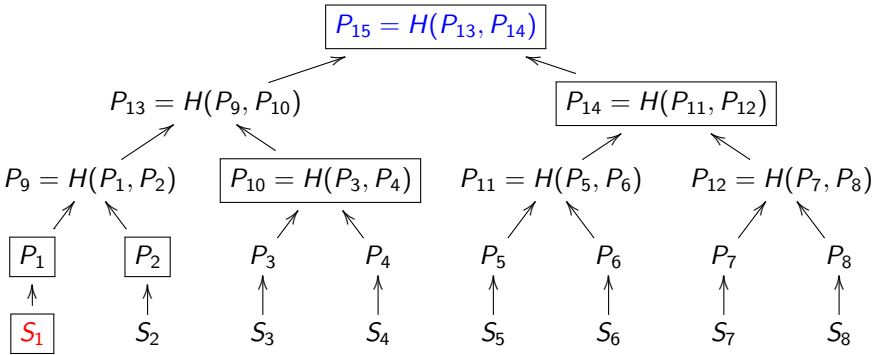
Hash 8 one-time public keys into a single Merkle public key P_{15}.



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

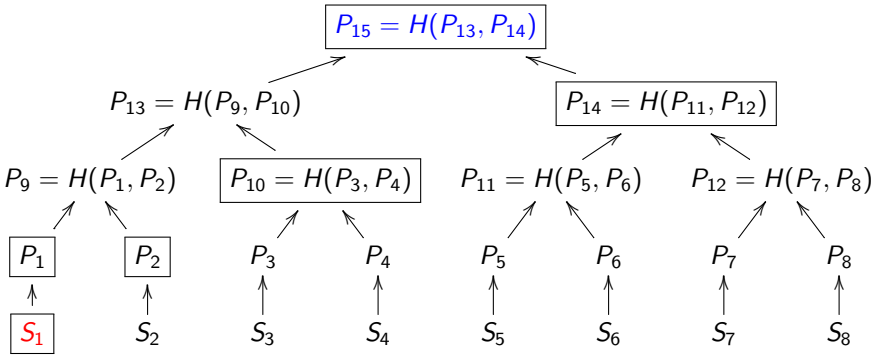
Signature in 8-time Merkle hash tree

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.

Considerations about 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.

Considerations about 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 to recompute one-time signing keys.

Considerations about Merkle's scheme

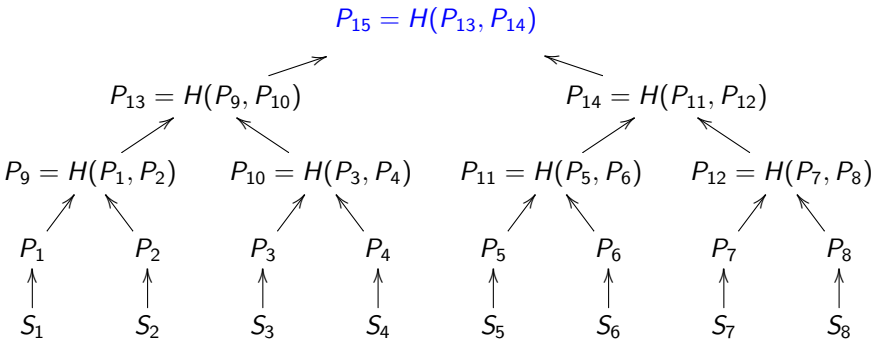
- Each key is good only for fixed number of messages, typically 2^n.
- The public key is very short: just one hash output.
  But each signature contains n public keys along with the one-time signature.
- Computing the public key requires computing and storing 2^n one-time signature keys.
- Can trade 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 to recompute one-time signing keys.
- 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.
  The standardized schemes XMSS and LMS are built this way.

Considerations about Merkle's scheme

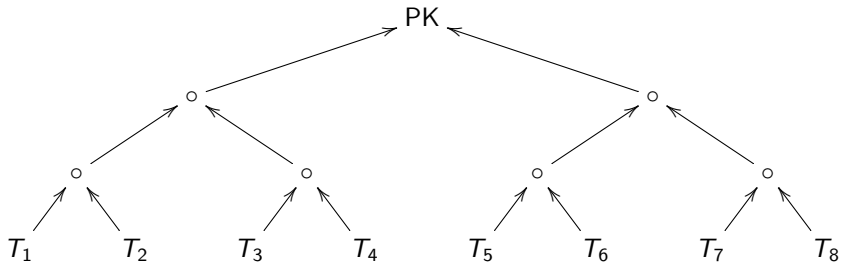
- Each key is good only for fixed number of messages, typically 2^n.
- The public key is very short: just one hash output.
  But each signature contains n public keys along with the one-time signature.
- Computing the public key requires computing and storing 2^n one-time signature keys.
- Can trade 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 to recompute one-time signing keys.
- 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.
  The standardized schemes XMSS and LMS are built this way.
- Do not forget: these are stateful schemes, you need to be able to count.

Stateless signatures

Trees of Merkle trees

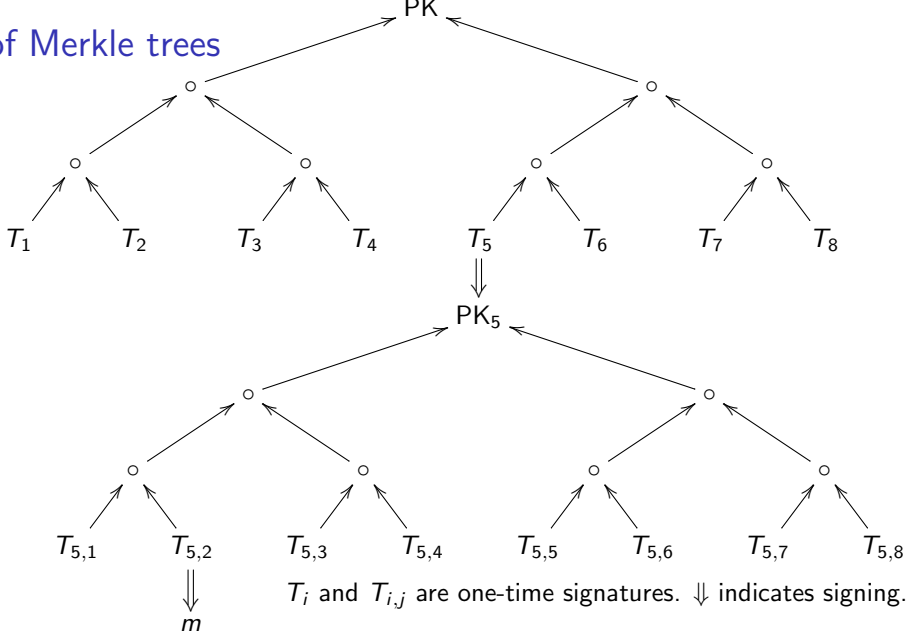


Trees of Merkle trees



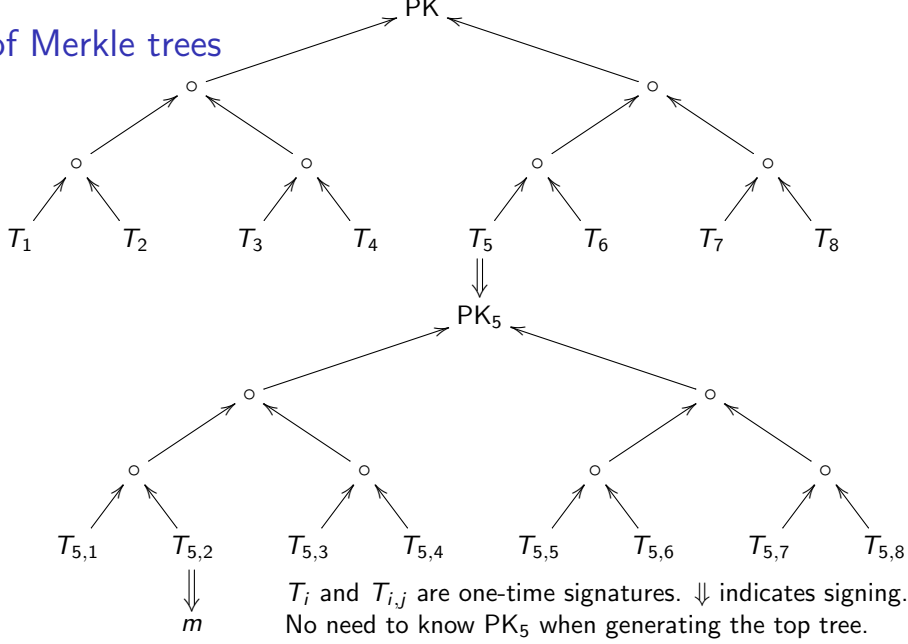
T_i are one-time signatures.
$\uparrow$ indicates input to hash function.

Trees of Merkle trees



T_i and $T_{i,j}$ are one-time signatures. $\Downarrow$ indicates signing.

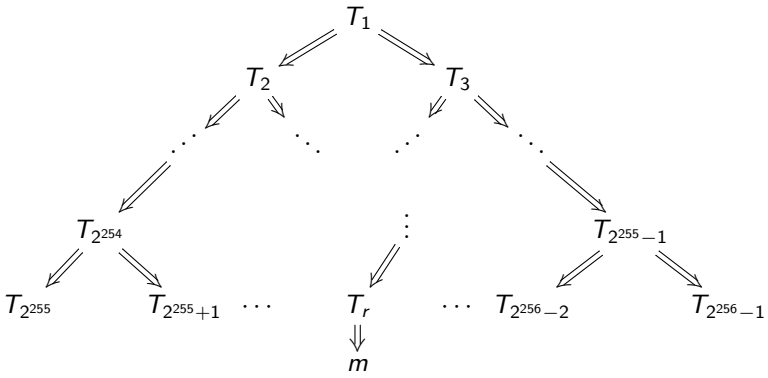
Trees of Merkle trees



T_i and $T_{i,j}$ are one-time signatures. $\Downarrow$ indicates signing.
No need to know PK_5 when generating the top tree.

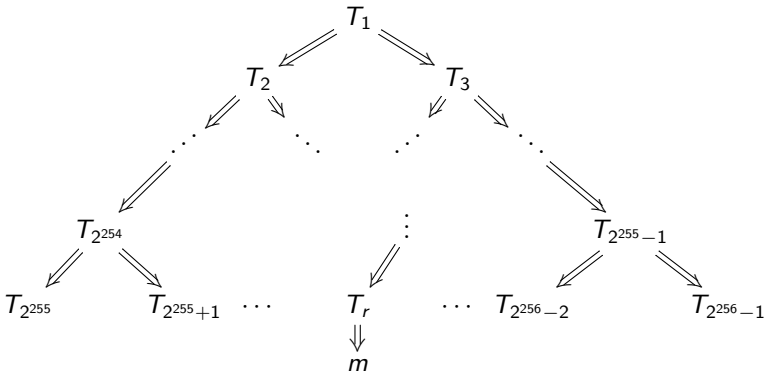
Huge trees (1987 Goldreich), keys on demand (Levin)

Signer chooses random $r \in \{2^{255}, 2^{255}+1, \ldots, 2^{256}-1\}$,
uses one-time public key T_r to sign message;
uses one-time public key T_i to sign (T_{2i}, T_{2i+1}) for $i < 2^{255}$.
Generates ith secret key as $H_k(i)$ where k is master secret.



Huge trees (1987 Goldreich), keys on demand (Levin)

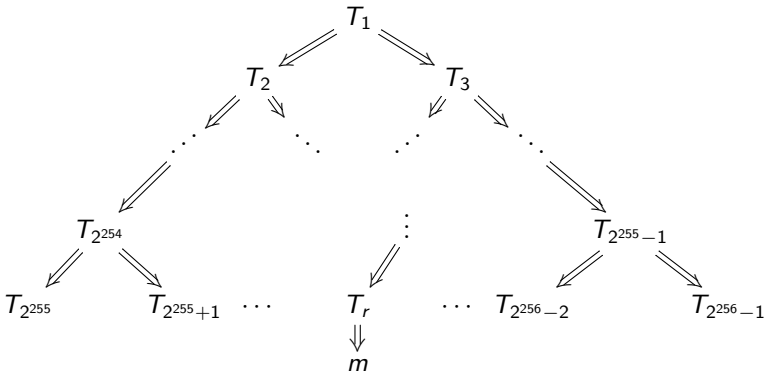
Signer chooses random $r \in \{2^{255}, 2^{255}+1, \ldots, 2^{256}-1\}$,
uses one-time public key T_r to sign message;
uses one-time public key T_i to sign (T_{2i}, T_{2i+1}) for $i < 2^{255}$.
Generates ith secret key as $H_k(i)$ where k is master secret.



T_i for small i gets used repeatedly (each time an m falls in that sub-tree)

Huge trees (1987 Goldreich), keys on demand (Levin)

Signer chooses random $r \in \{2^{255}, 2^{255}+1, \ldots, 2^{256}-1\}$,
uses one-time public key T_r to sign message;
uses one-time public key T_i to sign (T_{2i}, T_{2i+1}) for $i < 2^{255}$.
Generates ith secret key as $H_k(i)$ where k is master secret.



T_i for small i gets used repeatedly (each time an m falls in that sub-tree)
but $(H_k(2i), H_k(2i+1))$ being deterministic means T_i signs the same value, so no break.

Use Goldreich to create stateless hash-based signatures

0.6 MB for hash-based Goldreich signature using
short-public-key Winternitz-16 one-time signatures.

Would dominate traffic in typical applications,
and add user-visible latency on typical network connections.

Use Goldreich to create stateless hash-based signatures

0.6 MB for hash-based Goldreich signature using
short-public-key Winternitz-16 one-time signatures.

Would dominate traffic in typical applications,
and add user-visible latency on typical network connections.

Example:
Debian operating system is designed for frequent upgrades.
At least one new signature for each upgrade.
Typical upgrade: one package or just a few packages.
1.2 MB average package size.
0.08 MB median package size.

Use Goldreich to create stateless hash-based signatures

0.6 MB for hash-based Goldreich signature using
short-public-key Winternitz-16 one-time signatures.

Would dominate traffic in typical applications,
and add user-visible latency on typical network connections.

Example:
Debian operating system is designed for frequent upgrades.
At least one new signature for each upgrade.
Typical upgrade: one package or just a few packages.
1.2 MB average package size.
0.08 MB median package size.

Example:
HTTPS typically sends multiple signatures per page.
1.8 MB average web page in Alexa Top 1000000.

Ingredients of SPHINCS (and SPHINCS-256)

Drastically reduce tree height (to 60).

Replace one-time leaves with few-time leaves.

Optimize few-time signature size *plus* key size.
New few-time HORST, improving upon HORS
(see exercise sheet 4).

Use hyper-trees (12 layers), as in GMSS.

Use masks, as in XMSS and XMSS^{MT},
for standard-model security proofs.

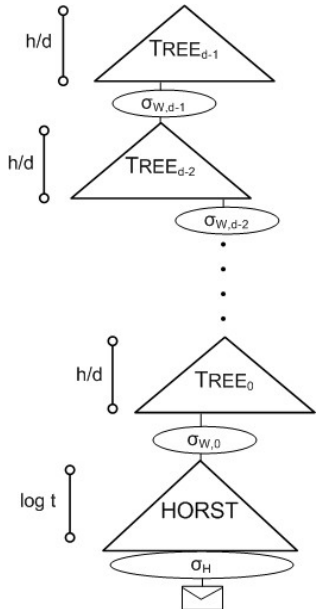
Optimize short-input (256-bit) hashing speed.
Use sponge hash (with ChaCha12 permutation).

Use fast stream cipher (again ChaCha12).

Vectorize hash software and cipher software.

See paper for details: `sphincs.cr.yp.to`

Updated version is NIST submission SPHINCS+
`https://sphincs.org/`.



Further information

- NISTs PQC competition.
- Quantum Threat Timeline vom Global Risk Institute, 2019; 2021 update.
- Status of quantum computer development (by German BSI).
- ENISA Study Post-Quantum Cryptography: Current state and quantum mitigation
- ENISA Study Post-Quantum Cryptography - Integration study
- YouTube Channel Tanja Lange: Post-quantum cryptography.
  Follow this in the Mastermath course "Selected Areas in Cryptography" next Spring.
- `https://2017.pqcrypto.org/school`: PQCRYPTO summer school
  with 21 lectures on video, slides, and exercises.
- `https://2017.pqcrypto.org/exec` and `https://pqcschool.org/index.html`:
  Executive school (less math, more perspective).
- `https://pqcrypto.org` our overview page.
- PQCrypto 2016, 2017, 2018, 2019, 2020, 2021, 2022 with many slides and videos online.
- `https://pqcrypto.eu.org`: PQCRYPTO EU Project.