Post-quantum cryptography

Peter Schwabe

Radboud University, The Netherlands

December 3, 2015

Santacrypt 2015, Prague, Czech Republic

Crypto in TLS

TLS_ECDH_ECDSA_WITH_NULL_SHA
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
TLS_SRP_SHA_WITH_AES_128_CBC_SHA
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA384
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA
TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA
TLS_ECDHE_RSA_WITH_NULL_SHA
TLS_DHE_DSS_WITH_AES_256_GCM_SHA384
TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256
TLS_ECDHE_ECDSA_WITH_NULL_SHA
TLS_SRP_SHA_WITH_AES_256_CBC_SHA
TLS_SRP_SHA_RSA_WITH_3DES_EDE_CBC_SHA
TLS_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA
TLS_PSK_WITH_AES_128_CBC_SHA256
TLS_DH_anon_WITH_AES_128_CBC_SHA256
TLS_ECDH_anon_WITH_3DES_EDE_CBC_SHA
TLS_DH_anon_WITH_CAMELLIA_128_CBC_SHA256
TLS_DH_DSS_WITH_AES_128_GCM_SHA256
TLS_DH_anon_WITH_AES_256_CBC_SHA256
TLS_SRP_SHA_RSA_WITH_AES_256_CBC_SHA
TLS_ECDH_RSA_WITH_NULL_SHA
TLS_ECDHE_PSK_WITH_AES_256_CBC_SHA384
TLS_ECDHE_PSK_WITH_3DES_EDE_CBC_SHA
TLS_DHE_DSS_EXPORT1024_WITH_RC4_56_SHA
TLS_RSA_WITH_SEED_CBC_SHA
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
TLS_DH_anon_WITH_CAMELLIA_128_CBC_SHA
TLS_DH_anon_WITH_AES_128_GCM_SHA256
TLS_PSK_WITH_RC4_128_SHA
TLS_NTRU_NSS_WITH_AES_256_CBC_SHA
TLS_DHE_DSS_WITH_SEED_CBC_SHA
TLS_RSA_WITH_HC_128_CBC_SHA
TLS_DH_anon_WITH_CAMELLIA_256_CBC_SHA
TLS_RSA_PSK_WITH_NULL_SHA
TLS_DHE_DSS_WITH_AES_256_CBC_SHA256
TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA
TLS_DHE_DSS_WITH_AES_128_CBC_SHA256
TLS_PSK_WITH_NULL_SHA256
TLS_NTRU_RSA_WITH_AES_128_CBC_SHA
TLS_DHE_PSK_WITH_AES_128_CBC_SHA256
TLS_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA256
TLS_DH_DSS_WITH_CAMELLIA_128_CBC_SHA
TLS_RSA_EXPORT1024_WITH_RC4_56_MD5
TLS_DH_RSA_WITH_CAMELLIA_128_CBC_SHA256
TLS_DH_DSS_WITH_CAMELLIA_256_CBC_SHA
TLS_DH_DSS_WITH_AES_128_CBC_SHA25
TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA
TLS_PSK_WITH_NULL_SHA384
TLS_ECDHE_PSK_WITH_AES_256_CBC_SHA
TLS_RSA_EXPORT1024_WITH_RC2_56_MD5
TLS_ECDH_RSA_WITH_AES_128_GCM_SHA256
TLS_RSA_WITH_CAMELLIA_128_CBC_SHA
TLS_KRB5_EXPORT_WITH_RC4_40_MD5
TLS_DHE_DSS_WITH_CAMELLIA_128_CBC_SHA256
TLS_RSA_WITH_AES_256_CBC_SHA
TLS_DHE_RSA_WITH_AES_128_CBC_SHA
TLS_DHE_PSK_WITH_RC4_128_SHA
TLS_PSK_WITH_AES_256_CBC_SHA
TLS_KRB5_EXPORT_WITH_DES_CBC_40_MD5
TLS_DHE_PSK_WITH_NULL_SHA384
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384
TLS_RSA_PSK_WITH_RC4_128_SHA
TLS_RSA_WITH_NULL_SHA256
TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
TLS_RSA_WITH_CAMELLIA_128_CBC_SHA256
TLS_ECDH_RSA_WITH_AES_256_GCM_SHA384
TLS_RSA_WITH_AES_128_CBC_SHA
SSL_FORTEZZA_KEA_WITH_RC4_128_SHA
SSL_CK_RC2_128_CBC_EXPORT40_WITH_MD5
TLS_DH_anon_WITH_DES_CBC_SHA
TLS_DH_DSS_WITH_AES_128_CBC_SHA
TLS_PSK_WITH_AES_128_CBC_SHA
TLS_ECDH_anon_WITH_NULL_SHA
TLS_DHE_DSS_WITH_CAMELLIA_128_CBC_SHA
TLS_DH_anon_WITH_RC4_128_MD5
TLS_PSK_WITH_NULL_SHA
TLS_DH_RSA_WITH_AES_256_GCM_SHA384
TLS_DH_DSS_EXPORT_WITH_DES40_CBC_SHA
TLS_DH_DSS_WITH_AES_256_CBC_SHA
TLS_DH_RSA_WITH_AES_256_CBC_SHA256
TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA256
TLS_KRB5_WITH_3DES_EDE_CBC_SHA
TLS_RSA_WITH_RABBIT_CBC_SHA
TLS_DHE_PSK_WITH_NULL_SHA
TLS_RSA_WITH_RC4_128_MD5
TLS_NTRU_RSA_WITH_RC4_128_SHA
TLS_SRP_SHA_WITH_3DES_EDE_CBC_SHA
TLS_DH_RSA_WITH_CAMELLIA_128_CBC_SHA
TLS_DH_anon_EXPORT_WITH_DES40_CBC_SHA
TLS_RSA_EXPORT_WITH_RC4_40_MD5
TLS_DHE_DSS_EXPORT1024_WITH_DES_CBC_SHA
SSL_CK_DES_64_CBC_WITH_MD5
TLS_RSA_WITH_DES_CBC_SHA
TLS_DHE_DSS_WITH_DES_CBC_SHA
TLS_RSA_WITH_RC4_128_SHA
SSL_CK_RC4_128_EXPORT40_WITH_MD5
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA
TLS_RSA_EXPORT1024_WITH_DES_CBC_SHA
TLS_ECDHE_ECDSA_WITH_RC4_128_SHA
SSL_CK_DES_192_EDE3_CBC_WITH_MD5
TLS_DH_RSA_WITH_DES_CBC_SHA
TLS_DHE_RSA_WITH_AES_256_CBC_SHA256
SSL_CK_RC2_128_CBC_WITH_MD5
TLS_PSK_WITH_AES_256_CBC_SHA384
TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA
TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
TLS_RSA_EXPORT_WITH_DES40_CBC_SHA
SSL_FORTEZZA_KEA_WITH_NULL_SHA
TLS_DHE_PSK_WITH_NULL_SHA256
SSL_CK_IDEA_128_CBC_WITH_MD5
SSL_CK_RC4_64_WITH_MD5
TLS_KRB5_WITH_3DES_EDE_CBC_MD5
TLS_DHE_RSA_WITH_AES_128_CBC_SHA256
TLS_NULL_WITH_NULL_NULL
TLS_KRB5_WITH_RC4_128_MD5
TLS_DH_RSA_EXPORT_WITH_DES40_CBC_SHA
TLS_RSA_WITH_3DES_EDE_CBC_SHA
TLS_DH_DSS_WITH_DES_CBC_SHA
TLS_KRB5_EXPORT_WITH_DES_CBC_40_SHA
TLS_RSA_PSK_WITH_AES_256_CBC_SHA
TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA
TLS_RSA_WITH_IDEA_CBC_SHA
TLS_DH_anon_WITH_3DES_EDE_CBC_SHA
TLS_KRB5_WITH_IDEA_CBC_MD5
TLS_KRB5_EXPORT_WITH_RC2_CBC_40_SHA
TLS_RSA_WITH_NULL_SHA
SSL_FORTEZZA_KEA_WITH_FORTEZZA_CBC_SHA
TLS_DH_DSS_WITH_AES_256_GCM_SHA384
TLS_RSA_WITH_NULL_MD5
TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA
TLS_DH_RSA_WITH_AES_128_GCM_SHA256
TLS_DH_anon_EXPORT_WITH_RC4_40_MD5
TLS_KRB5_WITH_RC4_128_SHA
TLS_KRB5_WITH_DES_CBC_MD5
SSL_RSA_FIPS_WITH_DES_CBC_SHA
TLS_DH_RSA_WITH_AES_256_CBC_SHA
TLS_KRB5_EXPORT_WITH_RC4_40_SHA
TLS_DHE_PSK_WITH_AES_128_CBC_SHA
TLS_KRB5_WITH_IDEA_CBC_SHA
TLS_DHE_DSS_WITH_AES_128_GCM_SHA256
TLS_DHE_RSA_WITH_DES_CBC_SHA
TLS_DH_anon_WITH_CAMELLIA_256_CBC_SHA256
TLS_RSA_WITH_AES_256_CBC_SHA256
TLS_DH_anon_WITH_AES_128_CBC_SHA
TLS_DH_RSA_WITH_AES_128_CBC_SHA
SSL_CK_RC4_128_WITH_MD5
TLS_KRB5_WITH_DES_CBC_SHA
TLS_RSA_PSK_WITH_3DES_EDE_CBC_SHA
TLS_RSA_PSK_WITH_AES_128_GCM_SHA256
TLS_KRB5_EXPORT_WITH_RC2_CBC_40_MD5
TLS_DH_anon_WITH_AES_256_CBC_SHA
TLS_PSK_WITH_3DES_EDE_CBC_SHA
TLS_DHE_DSS_WITH_RC4_128_SHA
TLS_RSA_WITH_AES_128_CBC_SHA256
SSL_RSA_FIPS_WITH_3DES_EDE_CBC_SHA
TLS_ECDHE_RSA_WITH_RC4_128_SHA
TLS_RSA_WITH_CAMELLIA_256_CBC_SHA
TLS_RSA_PSK_WITH_AES_256_GCM_SHA384
TLS_DHE_DSS_WITH_AES_128_CBC_SHA
TLS_RSA_PSK_WITH_AES_128_CBC_SHA256
TLS_DH_DSS_WITH_AES_256_CBC_SHA256
TLS_RSA_WITH_HC_128_CBC_MD5
TLS_SRP_SHA_RSA_WITH_AES_128_CBC_SHA
TLS_NTRU_NSS_WITH_AES_128_CBC_SHA
TLS_NTRU_NSS_WITH_RC4_128_SHA
TLS_PSK_WITH_AES_128_GCM_SHA256
TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA
TLS_DHE_DSS_WITH_AES_256_CBC_SHA
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
TLS_NTRU_RSA_WITH_3DES_EDE_CBC_SHA
TLS_DHE_RSA_WITH_SEED_CBC_SHA
TLS_DHE_PSK_WITH_AES_128_GCM_SHA256
TLS_NTRU_NSS_WITH_3DES_EDE_CBC_SHA
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
TLS_SRP_SHA_DSS_WITH_3DES_EDE_CBC_SHA
TLS_DH_RSA_WITH_AES_128_CBC_SHA256
TLS_DH_RSA_WITH_CAMELLIA_256_CBC_SHA
TLS_ECDH_ECDSA_WITH_RC4_128_SHA
TLS_RSA_PSK_WITH_NULL_SHA256
TLS_DH_DSS_WITH_SEED_CBC_SHA
TLS_RSA_PSK_WITH_AES_128_CBC_SHA
TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
TLS_RSA_PSK_WITH_AES_256_CBC_SHA384
TLS_DHE_PSK_WITH_3DES_EDE_CBC_SHA
TLS_RSA_WITH_AES_256_GCM_SHA384
TLS_DHE_RSA_WITH_AES_256_CBC_SHA
TLS_ECDH_anon_WITH_RC4_128_SHA
TLS_ECDH_RSA_WITH_RC4_128_SHA
TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA256
TLS_SRP_SHA_DSS_WITH_AES_256_CBC_SHA
TLS_DHE_PSK_WITH_AES_256_CBC_SHA
TLS_RSA_WITH_AES_128_GCM_SHA256
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256
TLS_RSA_PSK_WITH_NULL_SHA384
TLS_DH_DSS_WITH_CAMELLIA_256_CBC_SHA256
TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA
TLS_ECDHE_PSK_WITH_RC4_128_SHA
TLS_RSA_EXPORT1024_WITH_RC4_56_SHA
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
TLS_DH_DSS_WITH_CAMELLIA_128_CBC_SHA256
TLS_DH_anon_WITH_AES_256_GCM_SHA384
TLS_NTRU_RSA_WITH_AES_256_CBC_SHA
TLS_DHE_PSK_WITH_AES_256_GCM_SHA384
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
TLS_DH_RSA_WITH_SEED_CBC_SHA
TLS_ECDH_RSA_WITH_AES_128_CBC_SHA
TLS_ECDHE_PSK_WITH_NULL_SHA384
TLS_DH_RSA_WITH_CAMELLIA_256_CBC_SHA256
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
TLS_DH_ANON_WITH_SEED_CBC_SHA
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
TLS_ECDHE_PSK_WITH_NULL_SHA
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA
TLS_RSA_WITH_CAMELLIA_256_CBC_SHA256
TLS_PSK_WITH_AES_256_GCM_SHA384
TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
TLS_SRP_SHA_DSS_WITH_AES_128_CBC_SHA
TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA
TLS_ECDH_anon_WITH_AES_128_CBC_SHA
TLS_ECDH_anon_WITH_AES_256_CBC_SHA
TLS_DHE_PSK_WITH_AES_256_CBC_SHA384
TLS_ECDHE_PSK_WITH_NULL_SHA256
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256

Lots of choices to make...

- Some primitives are intentionally cryptographically weak (EXPORT)

Lots of choices to make. . .

- Some primitives are intentionally cryptographically weak (EXPORT)
- Some primitives are unintentionally cryptographically weak (RC4, MD5)

Lots of choices to make…

- Some primitives are intentionally cryptographically weak (EXPORT)
- Some primitives are unintentionally cryptographically weak (RC4, MD5)
- Some primitives are prone to implementation attacks (AES-CBC)

Lots of choices to make...

- Some primitives are intentionally cryptographically weak (EXPORT)
- Some primitives are unintentionally cryptographically weak (RC4, MD5)
- Some primitives are prone to implementation attacks (AES-CBC)
- Some primitives need very high-quality randomness ((EC-)DSA)

Lots of choices to make...

- Some primitives are intentionally cryptographically weak (EXPORT)
- Some primitives are unintentionally cryptographically weak (RC4, MD5)
- Some primitives are prone to implementation attacks (AES-CBC)
- Some primitives need very high-quality randomness ((EC-)DSA)
- What parameters are "secure enough"? 1024-bit RSA? 1024-bit DSA?

Lots of choices to make...

- Some primitives are intentionally cryptographically weak (EXPORT)
- Some primitives are unintentionally cryptographically weak (RC4, MD5)
- Some primitives are prone to implementation attacks (AES-CBC)
- Some primitives need very high-quality randomness ((EC-)DSA)
- What parameters are "secure enough"? 1024-bit RSA? 1024-bit DSA?

Very hard choices, easy to screw up!

Crypto in TLS that survives a “quantum attack”

[this slide intentionally left empty]

Quantum attacks

Definition

A *quantum attack* is an attack that is (partially) running on a quantum computer.

Quantum attacks

Definition

A *quantum attack* is an attack that is (partially) running on a quantum computer.

Should we be scared?

Largely accepted: A sufficiently large quantum computer does not exist (no, not even with the NSA, also not with DWAVE).

Quantum attacks

Definition

A *quantum attack* is an attack that is (partially) running on a quantum computer.

Should we be scared?

Largely accepted: A sufficiently large quantum computer does not exist (no, not even with the NSA, also not with DWAVE).

Should we be scared (part II)?

"In the past, people have said, maybe it's 50 years away, it's a dream, maybe it'll happen sometime. I used to think it was 50. Now I'm thinking like it's 15 or a little more. It's within reach. It's within our lifetime. It's going to happen."

—Mark Ketchen (IBM), Feb. 2012, about quantum computers

NSA's data center in Bluffdale

NSA's data center in Bluffdale

Estimated numbers

- Electricity consumption: 65 MW
- Energy bill: US$\$40,000,000$/year
- Storage: 3–12 EB

What will really be broken?

- RSA (encryption and signatures): dead (Shor)
- DSA, ElGamal, Schnorr etc.: dead (Shor)
- ECC (DH, ElGamal, signatures): dead (Shor)

What will really be broken?

- RSA (encryption and signatures): dead (Shor)
- DSA, ElGamal, Schnorr etc.: dead (Shor)
- ECC (DH, ElGamal, signatures): dead (Shor)
- Symmetric encryption: $\sqrt{}$-time for single-target key search (Grover)

What will really be broken?

- RSA (encryption and signatures): dead (Shor)
- DSA, ElGamal, Schnorr etc.: dead (Shor)
- ECC (DH, ElGamal, signatures): dead (Shor)
- Symmetric encryption: $\sqrt{}$-time for single-target key search (Grover)
- Hashes: $\sqrt{}$-time for single-target (second) preimages (Grover)
- Hashes: $\sqrt{}$-time for collision search (same as classical!)

PQCRYPTO

- Project funded by EU in Horizon 2020.
- Starting date 1 March 2015, runs for 3 years.
- 11 partners from academia and industry, TU/e is coordinator:

PQCRYPTO – aims and workpackages

Aims of PQCRYPTO

- Design a portfolio of high-security post-quantum public-key systems
- Provide efficient implementations of high-security post-quantum cryptography for a broad spectrum of real-world applications.

PQCRYPTO – aims and workpackages

Aims of PQCRYPTO

- Design a portfolio of high-security post-quantum public-key systems
- Provide efficient implementations of high-security post-quantum cryptography for a broad spectrum of real-world applications.

Technical work packages

- WP1: Post-quantum cryptography for small devices
  Leader: Tim Güneysu, co-leader: Peter Schwabe
- WP2: Post-quantum cryptography for the Internet
  Leader: Daniel J. Bernstein, co-leader: Bart Preneel
- WP3: Post-quantum cryptography for the cloud
  Leader: Nicolas Sendrier, co-leader: Lars Knudsen

PQCRYPTO – aims and workpackages

Aims of PQCRYPTO

- Design a portfolio of high-security post-quantum public-key systems
- Provide efficient implementations of high-security post-quantum cryptography for a broad spectrum of real-world applications.

Non-technical work packages

- WP4: Management and dissemination
  Leader: Tanja Lange
- WP5: Standardization
  Leader: Walter Fumy

POST-QUANTUM KEY EXCHANGE

A NEW HOPE

ERDEM ALKIM
LÉO DUCAS
THOMAS PÖPPELMANN
PETER SCHWABE

Ring-Learning-with-errors (RLWE)

- Let $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$
- Let χ be an *error distribution* on $\mathcal{R}_q$
- Let $\mathbf{s} \in \mathcal{R}_q$ be secret
- Attacker is given pairs $(\mathbf{a}, \mathbf{as} + \mathbf{e})$ with
  - $\mathbf{a}$ uniformly random from $\mathcal{R}_q$
  - $\mathbf{e}$ sampled from χ
- Task for the attacker: find $\mathbf{s}$

Ring-Learning-with-errors (RLWE)

- Let $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$
- Let χ be an *error distribution* on $\mathcal{R}_q$
- Let $\mathbf{s} \in \mathcal{R}_q$ be secret
- Attacker is given pairs $(\mathbf{a}, \mathbf{as} + \mathbf{e})$ with
  - $\mathbf{a}$ uniformly random from $\mathcal{R}_q$
  - $\mathbf{e}$ sampled from χ
- Task for the attacker: find $\mathbf{s}$
- Common choice for χ: discrete Gaussian

Ring-Learning-with-errors (RLWE)

- Let $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$
- Let χ be an *error distribution* on $\mathcal{R}_q$
- Let $\mathbf{s} \in \mathcal{R}_q$ be secret
- Attacker is given pairs $(\mathbf{a}, \mathbf{as} + \mathbf{e})$ with
  - $\mathbf{a}$ uniformly random from $\mathcal{R}_q$
  - $\mathbf{e}$ sampled from χ
- Task for the attacker: find $\mathbf{s}$
- Common choice for χ: discrete Gaussian
- Common "optimization" for protocols: fix $\mathbf{a}$ (more later)

Peikert's RLWE-based KEM

Parameters: q, n, χ		
KEM.Setup() :		
$\mathbf{a} \xleftarrow{\$} \mathcal{R}_q$		
Alice (server)		Bob (client)
KEM.Gen($\mathbf{a}$) :		KEM.Encaps($\mathbf{a}, \mathbf{b}$) :
$\mathbf{s}, \mathbf{e} \xleftarrow{\$} \chi$		$\mathbf{s}', \mathbf{e}', \mathbf{e}'' \xleftarrow{\$} \chi$
$\mathbf{b} \leftarrow \mathbf{a}\mathbf{s} + \mathbf{e}$	$\xrightarrow{\mathbf{b}}$	$\mathbf{u} \leftarrow \mathbf{a}\mathbf{s}' + \mathbf{e}'$
		$\mathbf{v} \leftarrow \mathbf{b}\mathbf{s}' + \mathbf{e}''$
		$\bar{\mathbf{v}} \xleftarrow{\$} \text{dbl}(\mathbf{v})$
KEM.Decaps($\mathbf{s}, (\mathbf{u}, \mathbf{v}')$) :	$\xleftarrow{\mathbf{u}, \mathbf{v}'}$	$\mathbf{v}' = \langle \bar{\mathbf{v}} \rangle_2$
$\mu \leftarrow \text{rec}(2\mathbf{u}\mathbf{s}, \mathbf{v}')$		$\mu \leftarrow \lfloor \bar{\mathbf{v}} \rceil_2$

Idea: $\mathbf{u}\mathbf{s} = \mathbf{a}\mathbf{s}\mathbf{s}' + \mathbf{e}'\mathbf{s} \approx \mathbf{a}\mathbf{s}\mathbf{s}' + \mathbf{e}\mathbf{s}' + \mathbf{e}'' = \mathbf{v}$

Use $\mathbf{v}'$ to resolve the problems from "$\approx$" (at least most of the time)

BCNS key exchange

- Bos, Costello, Naehrig, Stebila, IEEE S&P 2015:
  - Phrase the KEM as key exchange
  - Instantiate with concrete parameters
  - Integrate with OpenSSL $\rightarrow$ post-quantum TLS key exchange
  - Also: combined ECDH+RLWE key exchange

BCNS key exchange

- Bos, Costello, Naehrig, Stebila, IEEE S&P 2015:
  - Phrase the KEM as key exchange
  - Instantiate with concrete parameters
  - Integrate with OpenSSL $\rightarrow$ post-quantum TLS key exchange
  - Also: combined ECDH+RLWE key exchange
- Parameters chosen by BCNS:
  - $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$
  - $n = 1024$
  - $q = 2^{32} - 1$
  - $\chi = D_{\mathbb{Z},\sigma}$
  - $\sigma = 8\sqrt{2\pi} \approx 3.192$

BCNS key exchange

- Bos, Costello, Naehrig, Stebila, IEEE S&P 2015:
  - Phrase the KEM as key exchange
  - Instantiate with concrete parameters
  - Integrate with OpenSSL $\rightarrow$ post-quantum TLS key exchange
  - Also: combined ECDH+RLWE key exchange
- Parameters chosen by BCNS:
  - $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$
  - $n = 1024$
  - $q = 2^{32} - 1$
  - $\chi = D_{\mathbb{Z},\sigma}$
  - $\sigma = 8\sqrt{2\pi} \approx 3.192$
- Claimed security level: 128 bits *pre-quantum*

A new hope

- Improve failure analysis and error reconciliation
(smartly use the fact that we have 4 bits to encode one key bit)

A new hope

- Improve failure analysis and error reconciliation
  (smartly use the fact that we have 4 bits to encode one key bit)
- Drastically reduce q to $12289 < 2^{14}$

A new hope

- Improve failure analysis and error reconciliation
  (smartly use the fact that we have 4 bits to encode one key bit)
- Drastically reduce q to $12289 < 2^{14}$
- Analysis of *post-quantum* security

A new hope

- Improve failure analysis and error reconciliation
  (smartly use the fact that we have 4 bits to encode one key bit)
- Drastically reduce q to $12289 < 2^{14}$
- Analysis of *post-quantum* security
- Use centered binomial noise ψ_k ($\sum_{i=1}^{10} b_i - b'_i$ for $b_i, b'_i \in \{0, 1\}$)

A new hope

- Improve failure analysis and error reconciliation
  (smartly use the fact that we have 4 bits to encode one key bit)
- Drastically reduce q to $12289 < 2^{14}$
- Analysis of *post-quantum* security
- Use centered binomial noise ψ_k ($\sum_{i=1}^{10} b_i - b'_i$ for $b_i, b'_i \in \{0,1\}$)
- Choose a fresh parameter $\mathbf{a}$ for every protocol run

A new hope

- Improve failure analysis and error reconciliation
  (smartly use the fact that we have 4 bits to encode one key bit)
- Drastically reduce q to $12289 < 2^{14}$
- Analysis of *post-quantum* security
- Use centered binomial noise ψ_k ($\sum_{i=1}^{10} b_i - b'_i$ for $b_i, b'_i \in \{0, 1\}$)
- Choose a fresh parameter $\mathbf{a}$ for every protocol run
- Encode polynomials in NTT domain

A new hope

- Improve failure analysis and error reconciliation (smartly use the fact that we have 4 bits to encode one key bit)
- Drastically reduce q to $12289 < 2^{14}$
- Analysis of *post-quantum* security
- Use centered binomial noise ψ_k ($\sum_{i=1}^{10} b_i - b'_i$ for $b_i, b'_i \in \{0, 1\}$)
- Choose a fresh parameter $\mathbf{a}$ for every protocol run
- Encode polynomials in NTT domain
- Provide C reference and fast AVX2 implementation

A new hope – protocol

Parameters: $q = 12289 < 2^{14}, \; n = 1024$

Error distribution: ψ_{12}

Alice (server)		Bob (client)
$seed \xleftarrow{\$} \{0,1\}^{256}$		
$\mathbf{a} \leftarrow \text{Parse}(\text{SHAKE-128}(seed))$		
$\mathbf{s}, \mathbf{e} \xleftarrow{\$} \psi_8^n$		$\mathbf{s}', \mathbf{e}', \mathbf{e}'' \xleftarrow{\$} \psi_8^n$
$\mathbf{b} \leftarrow \mathbf{as} + \mathbf{e}$	$\xrightarrow{(\mathbf{b}, seed)}$	$\mathbf{a} \leftarrow \text{Parse}(\text{SHAKE-128}(seed))$
		$\mathbf{u} \leftarrow \mathbf{as}' + \mathbf{e}'$
		$\mathbf{v} \leftarrow \mathbf{bs}' + \mathbf{e}''$
$\mathbf{v}' \leftarrow \mathbf{us}$	$\xleftarrow{(\mathbf{u}, \mathbf{r})}$	$\mathbf{r} \xleftarrow{\$} \text{HelpRec}(\mathbf{v})$
$k \leftarrow \text{Rec}(\mathbf{v}', \mathbf{r})$		$k \leftarrow \text{Rec}(\mathbf{v}, \mathbf{r})$
$\mu \leftarrow \text{SHA3-256}(k)$		$\mu \leftarrow \text{SHA3-256}(k)$

Security analysis

- Consider RLWE instance as LWE instance
- Attack using BKZ
- BKZ uses SVP oracle in smaller dimension
- Consider only the cost of one call to that oracle ("core-SVP hardness")

Security analysis

- Consider RLWE instance as LWE instance
- Attack using BKZ
- BKZ uses SVP oracle in smaller dimension
- Consider only the cost of one call to that oracle (“core-SVP hardness”)
- Consider quantum sieve as SVP oracle
  - Best-known quantum cost (BKC): $2^{0.268n}$
  - Best-plausible quantum cost (BPC): $2^{0.2075n}$

Security analysis

- Consider RLWE instance as LWE instance
- Attack using BKZ
- BKZ uses SVP oracle in smaller dimension
- Consider only the cost of one call to that oracle ("core-SVP hardness")
- Consider quantum sieve as SVP oracle
  - Best-known quantum cost (BKC): $2^{0.268n}$
  - Best-plausible quantum cost (BPC): $2^{0.2075n}$
- Primal attack: unique-SVP from LWE; solve using BKZ

Security analysis

- Consider RLWE instance as LWE instance
- Attack using BKZ
- BKZ uses SVP oracle in smaller dimension
- Consider only the cost of one call to that oracle ("core-SVP hardness")
- Consider quantum sieve as SVP oracle
  - Best-known quantum cost (BKC): $2^{0.268n}$
  - Best-plausible quantum cost (BPC): $2^{0.2075n}$
- Primal attack: unique-SVP from LWE; solve using BKZ
- Dual attack: find short vector in dual lattice
- Length determines complexity and attacker's advantage ϵ

Post-quantum security

BCNS proposal

Attack	BKZ block dim. b	$\log_2$(BKC)	$\log_2$(BPC)
Primal	294	78	61
Dual ($\epsilon = 2^{-128}$)	230	62	48
Dual ($\epsilon = 1/2$)	331	89	69

A new hope

Attack	BKZ block dim. b	$\log_2$(BKC)	$\log_2$(BPC)
Primal	886	237	183
Dual ($\epsilon = 2^{-128}$)	658	176	136
Dual ($\epsilon = 1/2$)	1380	370	286

Against all authority

- Remember the optimization of fixed $\mathbf{a}$?
- What if $\mathbf{a}$ is backdoored?
- Parameter-generating authority can break key exchange
- "Solution": Nothing-up-my-sleeves (involves endless dicussion!)

Against all authority

- Remember the optimization of fixed $\mathbf{a}$?
- What if $\mathbf{a}$ is backdoored?
- Parameter-generating authority can break key exchange
- "Solution": Nothing-up-my-sleeves (involves endless dicussion!)
- Even without backdoor:
  - Perform massive precomputation based on $\mathbf{a}$
  - Use precomputation to break *all* key exchanges
  - Infeasible today, but who knows...
  - Attack in the spirit of Logjam

Against all authority

- Remember the optimization of fixed $\mathbf{a}$?
- What if $\mathbf{a}$ is backdoored?
- Parameter-generating authority can break key exchange
- "Solution": Nothing-up-my-sleeves (involves endless dicussion!)
- Even without backdoor:
  - Perform massive precomputation based on $\mathbf{a}$
  - Use precomputation to break *all* key exchanges
  - Infeasible today, but who knows...
  - Attack in the spirit of Logjam
- Solution in Newhope: Choose a fresh $\mathbf{a}$ every time
- Use SHAKE-128 to expand a 32-byte seed

Against all authority

- Remember the optimization of fixed $\mathbf{a}$?
- What if $\mathbf{a}$ is backdoored?
- Parameter-generating authority can break key exchange
- "Solution": Nothing-up-my-sleeves (involves endless dicussion!)
- Even without backdoor:
  - Perform massive precomputation based on $\mathbf{a}$
  - Use precomputation to break *all* key exchanges
  - Infeasible today, but who knows...
  - Attack in the spirit of Logjam
- Solution in Newhope: Choose a fresh $\mathbf{a}$ every time
- Use SHAKE-128 to expand a 32-byte seed
- Server can cache $\mathbf{a}$ for some time (e.g., 1h)

Implementation

- Very fast multiplication in $\mathcal{R}_q$: use NTT
- Define message format:
  - Send polynomials in NTT domain
  - Eliminate half of the required NTTs

The protocol revisited

Parameters: $q = 12289 < 2^{14}$, $n = 1024$

Error distribution: ψ_8

Alice (server)		**Bob (client)**
$seed \xleftarrow{\$} \{0, \ldots, 255\}^{32}$		
$\text{Parse}(\text{SHAKE-128}(seed))$		
$\mathbf{s}, \mathbf{e} \xleftarrow{\$} \psi_8^n$		$\mathbf{s}', \mathbf{e}', \mathbf{e}'' \xleftarrow{\$} \psi_8^n$
$\mathbf{b} \leftarrow \mathbf{a} \circ \text{NTT}(\mathbf{s}) + \text{NTT}(\mathbf{e})$	$\xrightarrow[\text{2048Bytes}]{m_a = \text{encodeA}(\mathbf{b}, seed)}$	$(\mathbf{b}, seed) \leftarrow \text{decodeA}(m_a)$
		$\mathbf{a} \leftarrow \text{Parse}(\text{SHAKE-128}(seed))$
		$\mathbf{t} \leftarrow \text{NTT}(\mathbf{s}')$
		$\mathbf{u} \leftarrow \mathbf{a} \circ \mathbf{t} + \text{NTT}(\mathbf{e}')$
		$\mathbf{v} \leftarrow \text{NTT}^{-1}(\mathbf{b} \circ \mathbf{t} + \text{NTT}(\mathbf{e}''))$
$(\mathbf{u}, \mathbf{r}) \leftarrow \text{decodeB}(m_b)$	$\xleftarrow[\text{2048 Bytes}]{m_b = \text{encodeB}(\mathbf{u}, \mathbf{r})}$	$\mathbf{r} \xleftarrow{\$} \text{HelpRec}(\mathbf{v})$
$\mathbf{v}' \leftarrow \text{NTT}^{-1}(\mathbf{u} \circ \mathbf{s})$		$k \leftarrow \text{Rec}(\mathbf{v}, \mathbf{r})$
$k \leftarrow \text{Rec}(\mathbf{v}', \mathbf{r})$		$\mu \leftarrow \text{SHA3-256}(k)$
$\mu \leftarrow \text{SHA3-256}(k)$		

Implementation

- Very fast multiplication in $\mathcal{R}_q$: use NTT
- Define message format:
  - Send polynomials in NTT domain
  - Eliminate half of the required NTTs
- C reference implementation:
  - Arithmetic on 16-bit and 32-bit integers
  - No division (/) or modulo (%) operator
  - Use Montgomery reductions inside NTT
  - Use ChaCha20 for noise sampling

Implementation

- Very fast multiplication in $\mathcal{R}_q$: use NTT
- Define message format:
  - Send polynomials in NTT domain
  - Eliminate half of the required NTTs
- C reference implementation:
  - Arithmetic on 16-bit and 32-bit integers
  - No division (/) or modulo (%) operator
  - Use Montgomery reductions inside NTT
  - Use ChaCha20 for noise sampling
- AVX2 implementation:
  - Speed up NTT using vectorized `double` arithmetic
  - Use AES-256 for noise sampling
  - Use AVX2 for centered binomial
  - Use AVX2 for error reconciliation

Implementation

- Very fast multiplication in $\mathcal{R}_q$: use NTT
- Define message format:
  - Send polynomials in NTT domain
  - Eliminate half of the required NTTs
- C reference implementation:
  - Arithmetic on 16-bit and 32-bit integers
  - No division (/) or modulo (%) operator
  - Use Montgomery reductions inside NTT
  - Use ChaCha20 for noise sampling
- AVX2 implementation:
  - Speed up NTT using vectorized `double` arithmetic
  - Use AES-256 for noise sampling
  - Use AVX2 for centered binomial
  - Use AVX2 for error reconciliation
- Microcontroller implementation (ongoing):
  - Cortex-M0
  - Cortex-M4

Performance

	BCNS	Ours (C ref)	Ours (AVX2)
Key generation (server)	$\approx 2\,477\,958$	$265\,968$ ($265\,933$)	$107\,534$ ($107\,385$)
Key gen + shared key (client)	$\approx 3\,995\,977$	$380\,676$ ($380\,936$)	$126\,236$ ($126\,336$)
Shared key (server)	$\approx 481\,937$	$82\,312$	$22\,104$

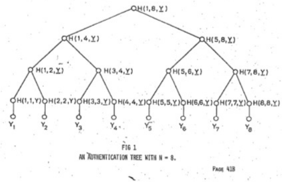
- Benchmarks on one core of an Intel i7-4770K (Haswell)
- BCNS benchmarks are derived from `openssl speed`
- Numbers in parantheses are average; all other numbers are median.
- Includes around $57\,000$ cycles for generation of $\mathbf{a}$ on each side

SPHINCS – stateless, practical, hash-based, incredibly nice, collision-resilient signatures

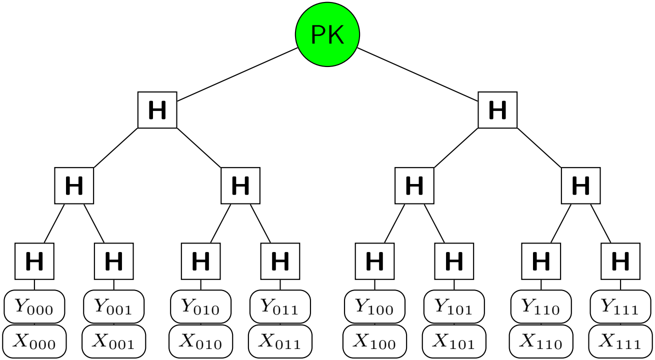
Daniel J. Bernstein
Daira Hopwood
Andreas Hülsing
Tanja Lange
Ruben Niederhagen
Louiza Papachristodoulou
Michael Schneider
Peter Schwabe
Zooko Wilcox-O'Hearn

Hash-based signatures

- Security relies only on secure hash function
  - Post-quantum
  - Reliable security estimates
- Fast (e.g., XMSS by Buchmann, Dahmen, Hülsing, 2011)
- Reasonably small keys, small signatures
- Stateful

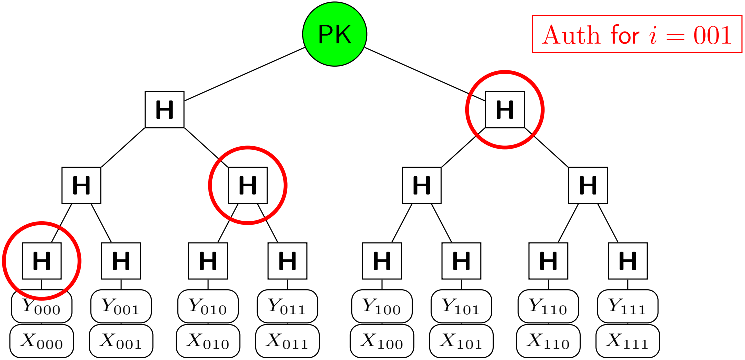


Merkle Trees



- Merkle, 1979: Leverage one-time signatures to multiple messages
- Binary hash tree on top of OTS public keys

Merkle Trees



- Use OTS keys sequentially
- $\mathrm{SIG} = (i, \mathrm{sign}(M, X_i), Y_i, \mathrm{Auth})$

About the state

- Used for *security*:
  Stores index $i \Rightarrow$ Prevents using one-time keys twice.
- Used for *efficiency*:
  Stores intermediate results for fast Auth computation.

About the state

- Used for *security*:
  Stores index $i \Rightarrow$ Prevents using one-time keys twice.
- Used for *efficiency*:
  Stores intermediate results for fast Auth computation.
- Problems:
  - Load-balancing
  - Multi-threading
  - Backups
  - Virtual-machine images
  - ...

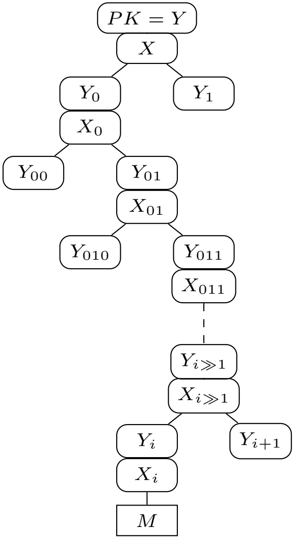
About the state

- Used for *security*:
  Stores index $i \Rightarrow$ Prevents using one-time keys twice.
- Used for *efficiency*:
  Stores intermediate results for fast Auth computation.
- Problems:
  - Load-balancing
  - Multi-threading
  - Backups
  - Virtual-machine images
  - ...
- This is not even compatible with the *definition* of cryptographic signatures
- "Huge foot-cannon" (Adam Langley, Google)

ELIMINATE THE STATE

Stateless hash-based signatures

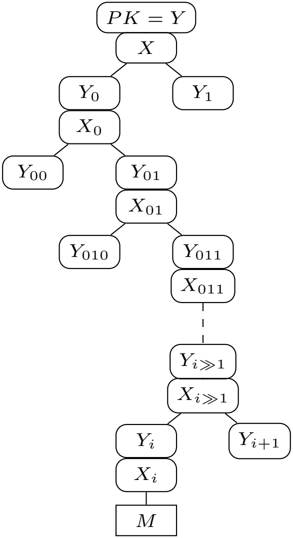
Goldreich's approach: Security parameter $\lambda = 128$
Use binary tree as in Merkle, but...



Stateless hash-based signatures

Goldreich's approach: Security parameter $\lambda = 128$
Use binary tree as in Merkle, but...

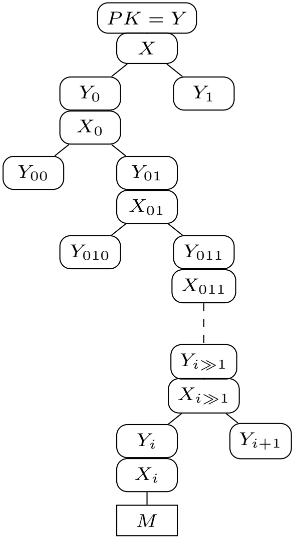
- For security
  - pick index i *at random*;
  - requires huge tree to avoid index collisions (e.g., height $h = 2\lambda = 256$).



Stateless hash-based signatures

Goldreich's approach: 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 $h = 2\lambda = 256$).
- For efficiency:
  - use binary *certification tree* of OTS;
  - all OTS secret keys are generated pseudorandomly.



It works, but signatures are painfully long

- 0.6 MB for 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.

It works, but signatures are painfully long

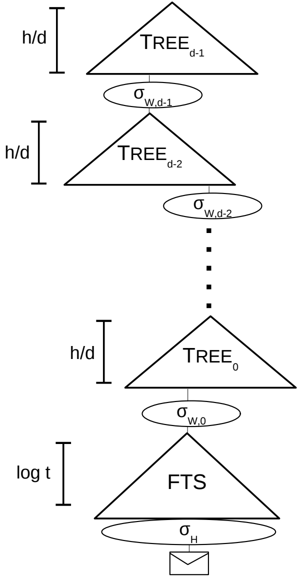
- 0.6 MB for Goldreich signature using short-public-key Winternitz-16 one-time signatures.
- Would dominate traffic in typical applications, and add user-visible latency on typical network connections.
- Example:
  - Debian operating system is designed for frequent upgrades.
  - At least one new signature for each upgrade.
  - Typical upgrade: one package or just a few packages.
  - 1.2 MB average package size.
  - 0.08 MB median package size.

It works, but signatures are painfully long

- 0.6 MB for 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.

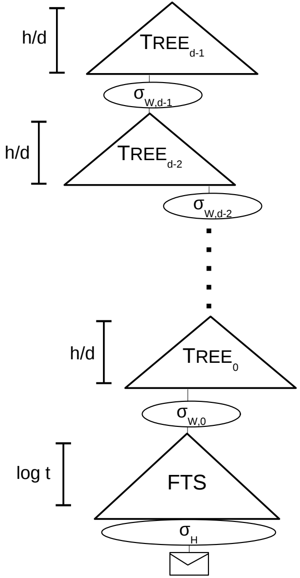
The SPHINCS approach

- Use a "hyper-tree" of total height h
- Parameter $d \geq 1$, such that $d \mid h$
- Each (Merkle) tree has height h/d
- (h/d)-ary certification tree

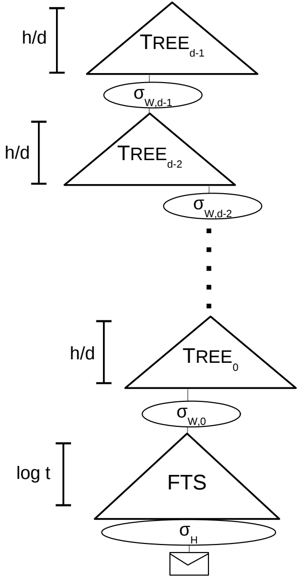


The SPHINCS approach

- Pick index (pseudo-)randomly
- Messages signed with *few-time* signature scheme
- Significantly reduce total tree height
- Require $\Pr[\text{r-times Coll}] \cdot \Pr[\text{Forgery after r signatures}] = \text{negl}(n)$



The SPHINCS approach



- Designed to be collision-resilient
- Trees: MSS-SPR trees
- OTS: WOTS$^+$
- FTS: HORST (HORS with tree)

SPHINCS-256

- Designed for 128 bits of post-quantum security
  (**yes, we did the analysis!**)
- 12 trees of height 5 each

SPHINCS-256

- Designed for 128 bits of post-quantum security (**yes, we did the analysis!**)
- 12 trees of height 5 each
- $n = 256$ bit hashes in WOTS and HORST
- Winternitz paramter $w = 16$
- HORST with 2^{16} expanded-secret-key chunks (total: 2 MB)

SPHINCS-256

- Designed for 128 bits of post-quantum security
  (**yes, we did the analysis!**)
- 12 trees of height 5 each
- $n = 256$ bit hashes in WOTS and HORST
- Winternitz paramter $w = 16$
- HORST with 2^{16} expanded-secret-key chunks (total: 2 MB)
- $m = 512$ bit message hash (BLAKE-512)
- ChaCha12 as PRG

Cost of SPHINCS-256 signing

- Three main componenents:
  - PRG for HORST secret-key expansion to 2 MB
  - Hashing in WOTS and HORS public-key generation:
    $F : \{0,1\}^{256} \to \{0,1\}^{256}$
  - Hashing in trees (mainly HORST public-key):
    $H : \{0,1\}^{512} \to \{0,1\}^{256}$
- Overall: $451\,456$ invocations of F, $91\,251$ invocations of H

Cost of SPHINCS-256 signing

- Three main componenents:
  - PRG for HORST secret-key expansion to 2 MB
  - Hashing in WOTS and HORS public-key generation: $F : \{0,1\}^{256} \to \{0,1\}^{256}$
  - Hashing in trees (mainly HORST public-key): $H : \{0,1\}^{512} \to \{0,1\}^{256}$
- Overall: $451\,456$ invocations of F, $91\,251$ invocations of H
- Full hash function would be overkill for F and H
- Construction in SPHINCS-256:
  - $F(M_1) = \mathsf{Chop}_{256}(\pi(M_1||C))$
  - $H(M_1||M_2) = \mathsf{Chop}_{256}(\pi(\pi(M_1||C) \oplus (M_2||0^{256})))$

Cost of SPHINCS-256 signing

- Three main componenents:
  - PRG for HORST secret-key expansion to 2 MB
  - Hashing in WOTS and HORS public-key generation:
    $F : \{0,1\}^{256} \to \{0,1\}^{256}$
  - Hashing in trees (mainly HORST public-key):
    $H : \{0,1\}^{512} \to \{0,1\}^{256}$
- Overall: $451\,456$ invocations of F, $91\,251$ invocations of H
- Full hash function would be overkill for F and H
- Construction in SPHINCS-256:
  - $F(M_1) = \mathsf{Chop}_{256}(\pi(M_1||C))$
  - $H(M_1||M_2) = \mathsf{Chop}_{256}(\pi(\pi(M_1||C) \oplus (M_2||0^{256})))$
- Use fast ChaCha12 permutation for π
- All building blocks (PRG, message hash, H, F) built from very similar permutations

SPHINCS-256 speed and sizes

SPHINCS-256 sizes

- 0.041 MB signature ($\approx 15\times$ smaller than Goldreich!)
- 0.001 MB public key
- 0.001 MB private key

SPHINCS-256 speed and sizes

SPHINCS-256 sizes

- 0.041 MB signature ($\approx 15\times$ smaller than Goldreich!)
- 0.001 MB public key
- 0.001 MB private key

High-speed implementation

- Target Intel Haswell with 256-bit AVX2 vector instructions
- Use $8\times$ parallel hashing, vectorize on high level
- ≈ 1.6 cycles/byte for H and F

SPHINCS-256 speed and sizes

SPHINCS-256 sizes

- 0.041 MB signature ($\approx 15\times$ smaller than Goldreich!)
- 0.001 MB public key
- 0.001 MB private key

High-speed implementation

- Target Intel Haswell with 256-bit AVX2 vector instructions
- Use $8\times$ parallel hashing, vectorize on high level
- ≈ 1.6 cycles/byte for H and F

SPHINCS-256 speed

- Signing: < 52 Mio. Haswell cycles (> 200 sigs/sec, 4 Core, 3GHz)
- Verification: < 1.5 Mio. Haswell cycles
- Keygen: < 3.3 Mio. Haswell cycles

Resources online

PQCRYPTO project: https://pqcrypto.eu.org

Newhope Paper: https://cryptojedi.org/papers/#newhope

Newhope Code: https://cryptojedi.org/crypto/#newhope

SPHINCS: https://sphincs.cr.yp.to/