Post-quantum cryptography

Peter Schwabe
Radboud University, The Netherlands

December 3, 2015

Santacrypt 2015, Prague, Czech Republic
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!
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”
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 D-WAVE).

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: $\sqrt{-1}$-time for single-target key search (Grover)
- Hashes: $\sqrt{-1}$-time for single-target (second) preimages (Grover)
- Hashes: $\sqrt{-1}$-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:

  - TU/e
  - BUNDDES DRUCKEREI
  - DTU
  - INRIA
  - KU LEUVEN
  - NXP
  - RUB
  - Radboud Universiteit
  - University of Haifa
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.
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
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
Ring-Learning-with-errors (RLWE)

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

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

Let $\chi$ be an error distribution on $\mathcal{R}_q$

Let $s \in \mathcal{R}_q$ be secret

Attacker is given pairs $(a, as + e)$ with
  - $a$ uniformly random from $\mathcal{R}_q$
  - $e$ sampled from $\chi$

Task for the attacker: find $s$

Common choice for $\chi$: discrete Gaussian

Common “optimization” for protocols: fix $a$ (more later)
Peikert’s RLWE-based KEM

<table>
<thead>
<tr>
<th>Parameters: $q, n, \chi$</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEM.Setup() :</td>
</tr>
<tr>
<td>$a \leftarrow R_q$</td>
</tr>
<tr>
<td>Alice (server)</td>
</tr>
<tr>
<td>KEM.Gen($a$) :</td>
</tr>
<tr>
<td>$s, e \leftarrow \chi$</td>
</tr>
<tr>
<td>$b \leftarrow as + e$</td>
</tr>
<tr>
<td>Bob (client)</td>
</tr>
<tr>
<td>KEM.Encaps($a, b$) :</td>
</tr>
<tr>
<td>$s', e', e'' \leftarrow \chi$</td>
</tr>
<tr>
<td>$u \leftarrow as' + e'$</td>
</tr>
<tr>
<td>$v \leftarrow bs' + e''$</td>
</tr>
<tr>
<td>$\bar{v} \leftarrow \text{dbl}(v)$</td>
</tr>
<tr>
<td>KEM.Decaps($s, (u, v')$) :</td>
</tr>
<tr>
<td>$\mu \leftarrow \text{rec}(2us, v')$</td>
</tr>
<tr>
<td>$v' = \langle \bar{v} \rangle_2$</td>
</tr>
<tr>
<td>$\mu \leftarrow [\bar{v}]_2$</td>
</tr>
</tbody>
</table>

Idea: $us = ass' + e's \approx ass' + es' + e'' = v$
Use $v'$ 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

Parameters chosen by BCNS:

- \(R_q = Z_q \left[\frac{X}{X^n+1}\right]\)
- \(n = 1024\)
- \(q = 2^{32} - 1\)
- \(\chi = D_{Z_{\sigma}}\)
- \(\sigma = 8\sqrt{2\pi} \approx 3.192\)

Claimed security level: 128 bits pre-quantum
**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:
  - \(\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)\)
  - \(n = 1024\)
  - \(q = 2^{32} - 1\)
  - \(\chi = D_{\mathbb{Z}_q,\sigma}\)
  - \(\sigma = 8\sqrt{2\pi} \approx 3.192\)

- Claimed security level: 128 bits pre-quantum
BCNS key exchange

- Bos, Costello, Naehrig, Stebila, IEEE S&P 2015:
  - Phrase the KEM as key exchange
  - Instantiate with concrete parameters
  - Integrate with OpenSSL → 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 \textit{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 $\psi_k (\sum_{i=1}^{10} b_i - b'_i$ 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 $\psi_{k} \left( \sum_{i=1}^{10} b_i - b'_i \right)$ for $b_i, b'_i \in \{0, 1\}$
- Choose a fresh parameter $a$ 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 $\psi_k \left( \sum_{i=1}^{10} b_i - b_i' \right.$ for $b_i, b_i' \in \{0, 1\} \left. \right)$
- Choose a fresh parameter $a$ 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 $\psi_k \left( \sum_{i=1}^{10} b_i - b'_i \right)$ for $b_i, b'_i \in \{0, 1\}$
- Choose a fresh parameter $a$ 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: \( \psi_{12} \)

<table>
<thead>
<tr>
<th>Alice (server)</th>
<th>Bob (client)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( seed \leftarrow {0, 1}^{256} )</td>
<td>( s', e', e'' \leftarrow \psi_{8}^{n} )</td>
</tr>
<tr>
<td>( a \leftarrow \text{Parse}(\text{SHAKE-128}(seed)) )</td>
<td>( a \leftarrow \text{Parse}(\text{SHAKE-128}(seed)) )</td>
</tr>
<tr>
<td>( s, e \leftarrow \psi_{8}^{n} )</td>
<td>( s', e', e'' \leftarrow \psi_{8}^{n} )</td>
</tr>
<tr>
<td>( b \leftarrow as + e )</td>
<td>( a \leftarrow \text{Parse}(\text{SHAKE-128}(seed)) )</td>
</tr>
<tr>
<td>( v' \leftarrow us )</td>
<td>( u \leftarrow as' + e' )</td>
</tr>
<tr>
<td>( k \leftarrow \text{Rec}(v', r) )</td>
<td>( v \leftarrow bs' + e'' )</td>
</tr>
<tr>
<td>( \mu \leftarrow \text{SHA3-256}(k) )</td>
<td>( k \leftarrow \text{Rec}(v, r) )</td>
</tr>
<tr>
<td>( \mu \leftarrow \text{SHA3-256}(k) )</td>
<td>( \mu \leftarrow \text{SHA3-256}(k) )</td>
</tr>
</tbody>
</table>
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 $\epsilon$
## Post-quantum security

### BCNS proposal

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

### A new hope

<table>
<thead>
<tr>
<th>Attack</th>
<th>BKZ block dim. $b$</th>
<th>$\log_2(BKC)$</th>
<th>$\log_2(BPC)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primal</td>
<td>886</td>
<td>237</td>
<td>183</td>
</tr>
<tr>
<td>Dual ($\epsilon = 2^{-128}$)</td>
<td>658</td>
<td>176</td>
<td>136</td>
</tr>
<tr>
<td>Dual ($\epsilon = 1/2$)</td>
<td>1380</td>
<td>370</td>
<td>286</td>
</tr>
</tbody>
</table>
Against all authority

- Remember the optimization of fixed $a$?
- What if $a$ is backdoored?
- Parameter-generating authority can break key exchange
- “Solution”: Nothing-up-my-sleeves (involves endless discussion!)
Against all authority

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

- Remember the optimization of fixed $a$?
- What if $a$ 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 $a$
  - Use precomputation to break all key exchanges
  - Infeasible today, but who knows... 
  - Attack in the spirit of Logjam
- Solution in Newhope: Choose a fresh $a$ every time
- Use SHAKE-128 to expand a 32-byte seed
- Server can cache $a$ 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: \( \psi_8 \)

### Alice (server)

\[
\text{seed} \leftarrow \{0, \ldots, 255\}^{32}
\]

Parse(SHAKE-128(\text{seed}))

\[
s, e \leftarrow \psi^n_8
\]

\[
b \leftarrow a \circ \text{NTT}(s) + \text{NTT}(e)
\]

\[
ma = \text{encodeA}(b, seed) \quad \xrightarrow{2048\text{Bytes}}
\]

\[
(u, r) \leftarrow \text{decodeB}(mb)
\]

\[
v' \leftarrow \text{NTT}^{-1}(u \circ s)
\]

\[
k \leftarrow \text{Rec}(v', r)
\]

\[
\mu \leftarrow \text{SHA3-256}(k)
\]

### Bob (client)

\[
s', e', e'' \leftarrow \psi^n_8
\]

\[
(b, seed) \leftarrow \text{decodeA}(ma)
\]

a \leftarrow \text{Parse}(\text{SHAKE-128}(seed))

\[
t \leftarrow \text{NTT}(s')
\]

\[
u \leftarrow a \circ t + \text{NTT}(e')
\]

\[
v \leftarrow \text{NTT}^{-1}(b \circ t + \text{NTT}(e''))
\]

\[
r \leftarrow \text{HelpRec}(v)
\]

\[
k \leftarrow \text{Rec}(v, r)
\]

\[
\mu \leftarrow \text{SHA3-256}(k)
\]
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
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

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

- 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 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, 1979: Leverage one-time signatures to multiple messages

Binary hash tree on top of OTS public keys
Merkle Trees

- Use OTS keys sequentially
- $\text{SIG} = (i, \text{sign}(M, X_i), Y_i, \text{Auth})$
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)
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.
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
  \[ \Pr[r\text{-times Coll}] \cdot \Pr[\text{Forgery after } r \text{ signatures}] = \text{negl}(n) \]
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 parameter $w = 16$
- 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 parameter $w = 16$
- 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 components:
  - PRG for HORST secret-key expansion to 2 MB
  - Hashing in WOTS and HORS public-key generation:
    \[ F : \{0, 1\}^{256} \rightarrow \{0, 1\}^{256} \]
  - Hashing in trees (mainly HORST public-key):
    \[ H : \{0, 1\}^{512} \rightarrow \{0, 1\}^{256} \]
- Overall: 451 456 invocations of \( F \), 91 251 invocations of \( H \)
Cost of SPHINCS-256 signing

- Three main components:
  - PRG for HORST secret-key expansion to 2 MB
  - Hashing in WOTS and HORS public-key generation:
    \[ F: \{0,1\}^{256} \rightarrow \{0,1\}^{256} \]
  - Hashing in trees (mainly HORST public-key):
    \[ H: \{0,1\}^{512} \rightarrow \{0,1\}^{256} \]
- 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:
  - \( F(M_1) = \text{Chop}_{256}(\pi(M_1||C)) \)
  - \( H(M_1||M_2) = \text{Chop}_{256}(\pi(\pi(M_1||C) \oplus (M_2||0^{256}))) \)
Cost of SPHINCS-256 signing

- Three main components:
  - PRG for HORST secret-key expansion to 2 MB
  - Hashing in WOTS and HORS public-key generation:
    \[ F : \{0, 1\}^{256} \rightarrow \{0, 1\}^{256} \]
  - Hashing in trees (mainly HORST public-key):
    \[ H : \{0, 1\}^{512} \rightarrow \{0, 1\}^{256} \]

- 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:
  - \[ F(M_1) = \text{Chop}_{256}(\pi(M_1||C)) \]
  - \[ H(M_1||M_2) = \text{Chop}_{256}(\pi(\pi(M_1||C) \oplus (M_2||0^{256}))) \]

- Use fast ChaCha12 permutation for \( \pi \)

- 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 (≈ 15× 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× 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
- $\approx 1.6$ 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/