Prototyping post-quantum and hybrid key exchange and authentication in TLS and SSH

Eric Crockett, Christian Paquin, Douglas Stebila

https://eprint.iacr.org/2019/858

https://github.com/awslabs/s2n

https://github.com/open-quantum-safe/

Overview

- Design considerations for hybrid modes of key exchange in general
- Case studies: designs and experimental outcomes
  - Key exchange:
    - TLS 1.2 in OpenSSL 1.0.2 and Amazon s2n
    - TLS 1.3 in OpenSSL 1.1.1
    - SSH v2 in OpenSSH 7.9
  - Authentication:
    - TLS 1.3 in OpenSSL 1.1.1
    - SSH v2 in OpenSSH 7.9

Design considerations
for hybrid modes of key exchange

Douglas Stebila, Scott Fluhrer, Shay Gueron. Design issues for hybrid key exchange in TLS 1.3. Internet-Draft. Internet Engineering Task Force, July 2019. https://tools.ietf.org/html/draft-stebila-tls-hybrid-design-01

Hybrid key exchange

- Multiple sources of interest in using multiple key exchange algorithms simultaneously as part of transition to post-quantum crypto
  - Several Internet-Drafts already:
    - TLS 1.2: Schanck, Whyte, Zhang 2016; Amazon 2019
    - TLS 1.3: Schanck, Stebila 2017; Whyte, Zhang, Fluhrer, Garcia-Morchon 2017; Kiefer, Kwiatkowski 2018; Stebila, Fluhrer, Gueron 2019
  - Experimental implementations: Google CECPQ1, CECPQ2; Open Quantum Safe; CECPQ2b; …
- Need PQ key exchange before we need PQ authentication because future quantum computers could retroactively decrypt, but not retroactively impersonate

Goals for hybridization

1. Backwards compatibility
    - Hybrid-aware client, hybrid-aware server
    - Hybrid-aware client, non-hybrid-aware server
    - Non-hybrid-aware client, hybrid-aware server
2. Low computational overhead
3. Low latency
4. No extra round trips
5. No duplicate information

Design options

- How to negotiate algorithms
- How to convey cryptographic data (public keys / ciphertexts)
- How to combine keying material

Negotiation: How many algorithms?

2

≥ 2

Negotiation: How to indicate which algorithms to use

Negotiate each algorithm individually

1. Standardize a name for each algorithm
2. Provide a data structure for conveying supported algorithms
3. Implement logic negotiating which combination

Negotiate pre-defined combinations of algorithms

1. Standardize a name for each desired combination

- Can use existing negotiation data structures and logic

Which option is preferred may depend on how many algorithms are ultimately standardized.

Conveying cryptographic data (public keys / ciphertexts)

1) Separate public keys

- For each supported algorithm, send each public key / ciphertext in its own parseable data structure

2) Concatenate public keys

- For each supported combination, concatenate its public keys / ciphertext into an opaque data structure

#1 requires protocol and implementation changes

#2 abstracts combinations into "just another single algorithm"

But #2 can also lead to sending duplicate values

- nistp256+bike1l1
- nistp256+sikep403
- nistp256+frodo640aes
- sikep403+frodo640aes

3x nistp256, 2x sikep403, 2x frodo640aes public keys

Combining keying material

Top requirement: needs to provide "robust" security:

- Final session key should be secure as long as at least one of the ingredient keys is unbroken
- (Most obvious techniques are fine, though with some subtleties; see Giacon et al. PKC 2018, Bindel et al. PQCrypto 2019, … .)

- XOR keys
- Concatenate keys and use directly
- Concatenate keys then apply a hash function / KDF
- Extend the protocol's "key schedule" with new stages for each key
- Insert the 2nd key into an unused spot in the protocol's key schedule

Emerging consensus?

- **Combining keying material**: concatenate keys then apply hash function / KDF

- **Number of algorithms**: 2 vs ≥ 2: no consensus
- **Negotiation**: negotiate algorithms separately versus in combination: no consensus
  - All(?) implementations to date have negotiated pre-defined combinations
- **Conveying public keys**: separately versus concatenated: no consensus
  - All(?) implementations to date have used concatenation

Key exchange case studies

Case study 1: TLS 1.2 in Amazon s2n

- Multi-level negotiation following TLS 1.2 design style:
  - Top-level ciphersuite with algorithm family: e.g. `TLS_`**`ECDHE_SIKE`**`_ECDSA_WITH_AES_256_GCM_SHA384`
  - Extensions used to negotiate parameterization within family:
    - 1 extension for which ECDH elliptic curve: nistp256, curve25519, …
    - 1 extension for which PQ parameterization: sikep403, sikep504, …
- Session key: concatenate session keys and apply KDF with public key/ciphertext as KDF label
- Experimental results: successfully implemented using nistp256+{bike1l1, sikep503}

Implementation base for rest of case studies

- Implementations from Open Quantum Safe project's liboqs library
  - Open-source C library collecting implementations of many round 2 KEMs and signature schemes – directly from contributors, from NIST submission packages, or via PQClean
  - https://github.com/open-quantum-safe
- Algorithms tested:
  - KEMs: 9 of 17 (BIKE round 1, FrodoKEM, Kyber, LEDAcrypt, NewHope, NTRU, NTS (1 variant), Saber, SIKE)
  - Signature schemes: 6 of 9 (Dilithium, MQDSS, Picnic, qTesla (round 1), Rainbow, SPHINCS+)

Case study 2: TLS 1.2 in OpenSSL 1.0.2
Case study 3: TLS 1.3 in OpenSSL 1.1.1
Case study 4: SSH v2 in OpenSSH 7.9

- Negotiate pairs of algorithms in pre-defined combinations
- Session key: concatenate session keys and use directly in key schedule

- Easy implementation, no change to negotiation logic

1st circle: PQ only
2nd circle: hybrid ECDH

● = success

◓ = fixable by changing implementation parameter

○ = would violate spec or otherwise unresolved error

† = algorithm on testing branch

	s2n (TLS 1.2)	OpenSSL 1.0.2 (TLS 1.2)	OpenSSL 1.1.1 (TLS 1.3)	OpenSSH
BIKE1-L1 (round 1)	– ●	● ●	● ●	● ●
BIKE1-L3 (round 1)	– –	● ●	● ●	● ●
BIKE1-L5 (round 1)	– –	● ●	● ●	● ●
BIKE2-L1 (round 1)	– –	● ●	● ●	● ●
BIKE2-L3 (round 1)	– –	● ●	● ●	● ●
BIKE2-L5 (round 1)	– –	● ●	● ●	● ●
BIKE3-L1 (round 1)	– –	● ●	● ●	● ●
BIKE3-L3 (round 1)	– –	● ●	● ●	● ●
BIKE3-L5 (round 1)	– –	● ●	● ●	● ●
FrodoKEM-640-AES	– –	● ●	● ●	● ●
FrodoKEM-640-SHAKE	– –	● ●	● ●	● ●
FrodoKEM-976-AES	– –	● ●	● ●	● ●
FrodoKEM-976-SHAKE	– –	● ●	● ●	● ●
FrodoKEM-1344-AES	– –	◓ ◓	◓ ◓	● ●
FrodoKEM-1344-SHAKE	– –	◓ ◓	◓ ◓	● ●
Kyber512	– –	● ●	● ●	● ●
Kyber768	– –	● ●	● ●	● ●
Kyber1024	– –	● ●	● ●	● ●
LEDAcrypt-KEM-LT-12†	– –	● ●	● ●	● ●
LEDAcrypt-KEM-LT-32†	– –	● ●	● ●	● ●
LEDAcrypt-KEM-LT-52†	– –	● ●	● ●	● ●
NewHope-512-CCA	– –	● ●	● ●	● ●
NewHope-1024-CCA	– –	● ●	● ●	● ●
NTRU-HPS-2048-509	– –	● ●	● ●	● ●
NTRU-HPS-2048-677	– –	● ●	● ●	● ●
NTRU-HPS-4096-821	– –	● ●	● ●	● ●
NTRU-HRSS-701	– –	● ●	● ●	● ●
NTS-KEM(12,64)†	– –	○ ○	○ ○	○ ○
LightSaber-KEM	– –	● ●	● ●	● ●
Saber-KEM	– –	● ●	● ●	● ●
FireSaber-KEM	– –	● ●	● ●	● ●
SIKEp503 (round 1)	– ●	– –	– –	– –
SIKEp434	– –	● ●	● ●	● ●
SIKEp503	– –	● ●	● ●	● ●
SIKEp610	– –	● ●	● ●	● ●
SIKEp751	– –	● ●	● ●	● ●

FrodoKEM 976, 1344

- OpenSSL 1.0.2 / TLS 1.2: too large for a pre-programmed buffer size, but easily fixed by increasing one buffer size
- OpenSSL 1.1.1 / TLS 1.3: same

NTS-KEM

- OpenSSL 1.0.2 / TLS 1.2: theoretically within spec's limitation of 2^{24} bytes, but buffer sizes that large caused failures we couldn't track down
- OpenSSL 1.1.1 / TLS 1.3: too large for spec (2^{16}-1 bytes)
- OpenSSH: theoretically within spec but not within RFC's "SHOULD", but couldn't resolve bugs

Authentication case studies

1st circle: PQ only
2nd circle: hybrid RSA

● = success

◒ = fixable by changing implementation parameter

○ = would violate spec or otherwise unresolved error

† = algorithm on testing branch

	OpenSSL 1.1.1 (TLS 1.3)
Dilithium-2	● ●
Dilithium-3	● ●
Dilithium-4	● ●
MQDSS-31-48	◒ ◒
MQDSS-31-64	◒ ◒
Picnic-L1-FS	◒ ◒
Picnic-L1-UR	◒ ◒
Picnic-L3-FS	○ ○
Picnic-L3-UR	○ ○
Picnic-L5-FS	○ ○
Picnic-L5-UR	○ ○
Picnic2-L1-FS	● ●
Picnic2-L3-FS	◒ ◒
Picnic2-L5-FS	◒ ◒
qTesla-I (round 1)	● ●
qTesla-III-size (round 1)	● ●
qTesla-III-speed (round 1)	● ●
Rainbow-Ia-Classic†	◒ ◒
Rainbow-Ia-Cyclic†	● ●
Rainbow-Ia-Cyclic-Compressed†	● ●
Rainbow-IIIc-Classic†	◒ ◒
Rainbow-IIIc-Cyclic†	◒ ◒
Rainbow-IIIc-Cyclic-Compressed†	◒ ◒
Rainbow-Vc-Classic†	◒ ◒
Rainbow-Vc-Cyclic†	◒ ◒
Rainbow-Vc-Cyclic-Compressed†	◒ ◒
SPHINCS+-{Haraka,SHA256,SHAKE256}-128f-{robust,simple}	◒ ◒
SPHINCS+-{Haraka,SHA256,SHAKE256}-128s-{robust,simple}	● ●
SPHINCS+-{Haraka,SHA256,SHAKE256}-192f-{robust,simple}	◒ ◒
SPHINCS+-{Haraka,SHA256,SHAKE256}-192s-{robust,simple}	◒ ◒
SPHINCS+-{Haraka,SHA256,SHAKE256}-256f-{robust,simple}	◒ ◒
SPHINCS+-{Haraka,SHA256,SHAKE256}-256s-{robust,simple}	◒ ◒

TLS 1.3:
- Max certificate size: 2^{24}-1
- Max signature size: 2^{16}-1

OpenSSL 1.1.1:
- Max certificate size: 102,400 bytes, but runtime enlargeable
- Max signature size: 2^{14}

1st circle: PQ only
2nd circle: hybrid RSA

● = success

◓ = fixable by changing implementation parameter

○ = would violate spec or otherwise unresolved error

† = algorithm on testing branch

	OpenSSL 1.1.1 (TLS 1.3)	OpenSSH
Dilithium-2	● ●	● ●
Dilithium-3	● ●	● ●
Dilithium-4	● ●	● ●
MQDSS-31-48	◓ ◓	● ●
MQDSS-31-64	◓ ◓	● ●
Picnic-L1-FS	◓ ◓	● ●
Picnic-L1-UR	◓ ◓	● ●
Picnic-L3-FS	○ ○	● ●
Picnic-L3-UR	○ ○	● ●
Picnic-L5-FS	○ ○	● ●
Picnic-L5-UR	○ ○	● ●
Picnic2-L1-FS	● ●	● ●
Picnic2-L3-FS	◓ ◓	● ●
Picnic2-L5-FS	◓ ◓	● ●
qTesla-I (round 1)	● ●	● ●
qTesla-III-size (round 1)	● ●	● ●
qTesla-III-speed (round 1)	● ●	● ●
Rainbow-Ia-Classic†	◓ ◓	◓ ◓
Rainbow-Ia-Cyclic†	● ●	● ●
Rainbow-Ia-Cyclic-Compressed†	● ●	● ●
Rainbow-IIIc-Classic†	◓ ◓	○ ○
Rainbow-IIIc-Cyclic†	◓ ◓	○ ○
Rainbow-IIIc-Cyclic-Compressed†	◓ ◓	○ ○
Rainbow-Vc-Classic†	◓ ◓	○ ○
Rainbow-Vc-Cyclic†	◓ ◓	○ ○
Rainbow-Vc-Cyclic-Compressed†	◓ ◓	○ ○
SPHINCS+-{Haraka,SHA256,SHAKE256}-128f-{robust,simple}	◓ ◓	● ●
SPHINCS+-{Haraka,SHA256,SHAKE256}-128s-{robust,simple}	● ●	● ●
SPHINCS+-{Haraka,SHA256,SHAKE256}-192f-{robust,simple}	◓ ◓	● ●
SPHINCS+-{Haraka,SHA256,SHAKE256}-192s-{robust,simple}	◓ ◓	● ●
SPHINCS+-{Haraka,SHA256,SHAKE256}-256f-{robust,simple}	◓ ◓	● ●
SPHINCS+-{Haraka,SHA256,SHAKE256}-256s-{robust,simple}	◓ ◓	● ●

OpenSSH maximum packet size: 2^{18}

Summary

Summary

- Several design choices for hybrid key exchange in network protocols on negotiation and transmitting public keys, no consensus

- Protocols have size constraints which prevent some schemes from being used

- Implementations may have additional size constraints which affect some schemes, which can be bypassed with varying degrees of success

Extensions and open questions

Remaining Round 2 candidates

- Welcome help in getting code into our framework – either directly into liboqs or via PQClean

Constraints in other parts of the protocol ecosystem

- Other client/server implementations
- Middle boxes

Performance

- Latency and throughput in lab conditions
- Latency in realistic network conditions à la [Lan18]

Use in applications

- Tested our OpenSSL experiment with Apache, nginx, links, OpenVPN, with reasonable success
- More work to do:
  S/MIME, more TLS clients, …

Acknowledgements

- Goutam Tamvada (University of Waterloo)
- Matthew Campagna, Shay Gueron, and Torben Hansen (Amazon Web Services); Christopher Wood; Michele Mosca and John Schanck (University of Waterloo)
- Open Quantum Safe project
  - Contributors: Nicholas Allen, Maxime Anvari, Mira Belenkiy, Ben Davies, Nir Drucker, Javad Doliskani, Vlad Gheorghiu, Shay Gueron, Torben Hansen, Andrew Hopkins, Kevin Kane, Karl Knopf, Tancrède Lepoint, Shravan Mishra, Alex Parent, Peter Schwabe, John Underhill, and Sebastian Verschoor; https://github.com/open-quantum-safe/liboqs/graphs/contributors
  - Financial support from Amazon Web Services, Tutte Institute for Mathematics and Computing
  - In-kind developer time from Amazon Web Services, Cisco Systems, evolutionQ, Microsoft Research
- PQClean (https://github.com/PQClean/PQClean)
  - Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Thom Wiggers; https://github.com/PQClean/PQClean/graphs/contributors
- Natural Sciences and Engineering Research Council (NSERC) of Canada Discovery grant RGPIN-2016-05146 and a NSERC Discovery Accelerator Supplement

Prototyping post-quantum and hybrid key exchange and authentication in TLS and SSH

Eric Crockett, Christian Paquin, Douglas Stebila

https://eprint.iacr.org/2019/858

https://github.com/awslabs/s2n

https://github.com/open-quantum-safe/