An Introduction to Post-Quantum Cryptography for Linux System Administrators

Presented by Marie Curie Ramirez and Atom Ramirez

Agenda

- What is Post Quantum Cryptography (PQC)
- What is Quantum Computing
- The PQC Algorithms
- Implementing PQC in SSH and HTTPS (TLS) services

Marie Curie Ramirez

UC Berkeley - Sophomore

Applied Mathematics and Data Science

Cal

WiCDS (Women in Computing and Data Science)
MPS Scholar Mentor
CalTeach

Atom Ramirez

UC Merced - Junior

Applied Mathematics, concentration in Computer Science

- SATAL Research Intern
- President of ACM
- Presented on PKI, Linux, Cybersecurity

What is Post Quantum Cryptography (PQC)?

“Quantum Safe”

cryptography focused on developing encryption algorithms that can run on today’s classical computers but withstand attacks from future quantum computers

“Quantum Resistant”

“Quantum Proof Encryption”

Why we need PQC

- Future proof - Quantum computers could break many public-key cryptosystems, which would compromise digital communication such as PKI, TLS, SSH, Encryption at rest
- Quantum Attacks - PQC aims to keep existing public key infrastructure secure in the future
- Compliance - Aligns with emerging cryptographic standards

Quantum Threat - Harvest Now, Decrypt Later

1

Data Collection

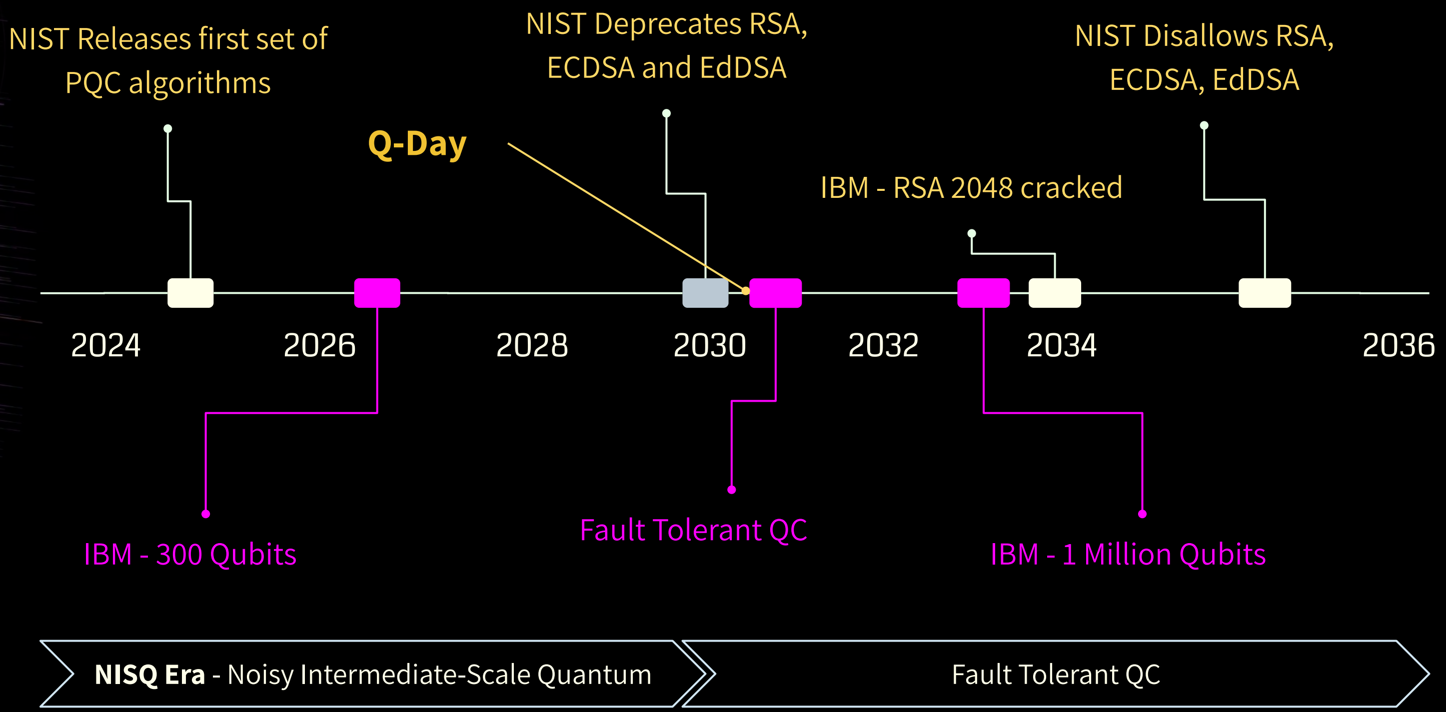
Attackers could be collecting encrypted data today using current encryption algorithms, such as RSA or ECC

2

Future Decryption

When quantum computers reach sufficient capabilities, attackers can use algorithms like Shor's to decrypt the collected data, revealing sensitive information that was previously protected by today's encryption methods.

The Quantum Threat Timeline



Post Quantum Cryptography ≠ Quantum Cryptography

Classical encryption algorithms, resistant to quantum computers

Encryption algorithms using Quantum Computers

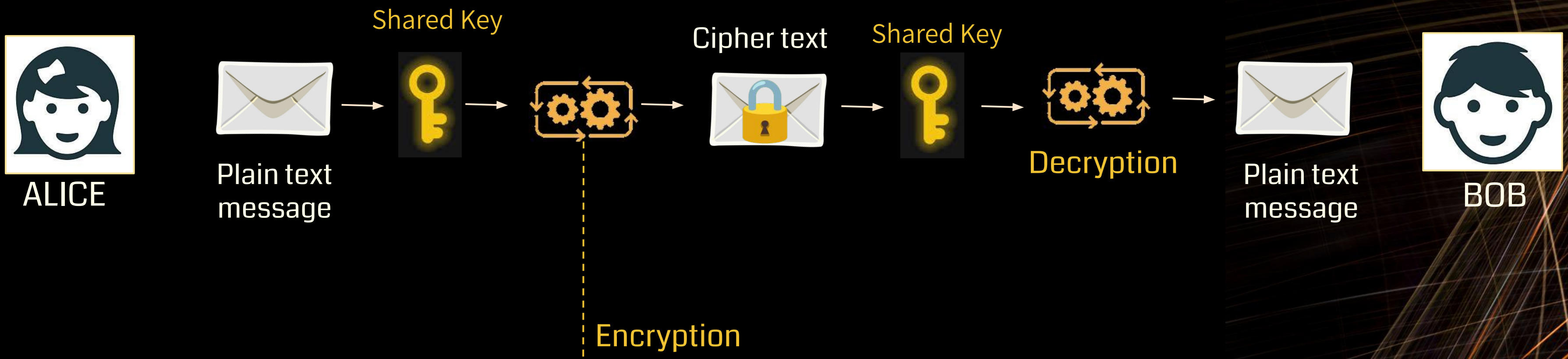
Cryptography 101

- Types of cryptographic categories
  - **Symmetric** - uses a single shared key for both encrypting and decrypting data
    - faster and more efficient than asymmetric encryption
    - typically used for bulk encryption / encrypting large amounts of data
  - **Asymmetric (Public Key Cryptography)** - uses two separate keys (public and private) where one key encrypts data and the other decrypts it
  - **Hashing functions** - is a one-way function that transforms data into a fixed-length string. Primarily used to verify data integrity rather than confidentiality

Cryptography 101: Symmetric Encryption

Alice wants to send a secret message to Bob

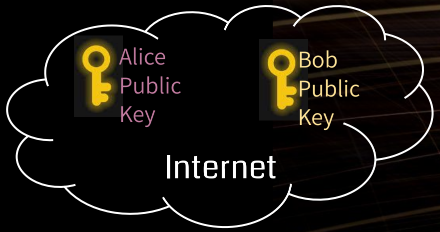
They exchange a shared secret key to decrypt and encrypt the message

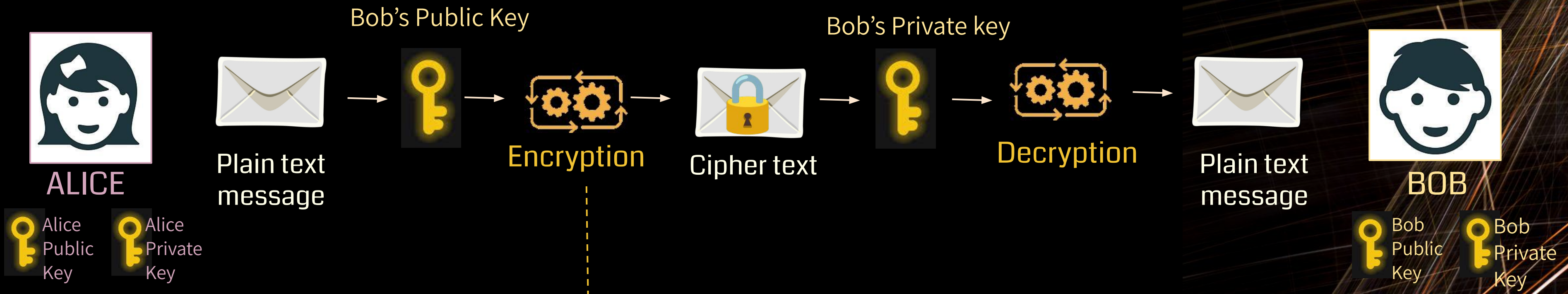


- AES (Advanced Encryption Standard) + key length (128, 192, 256)
- DES (Data Encryption Standard) - 3DES

Cryptography 101: Asymmetric Encryption

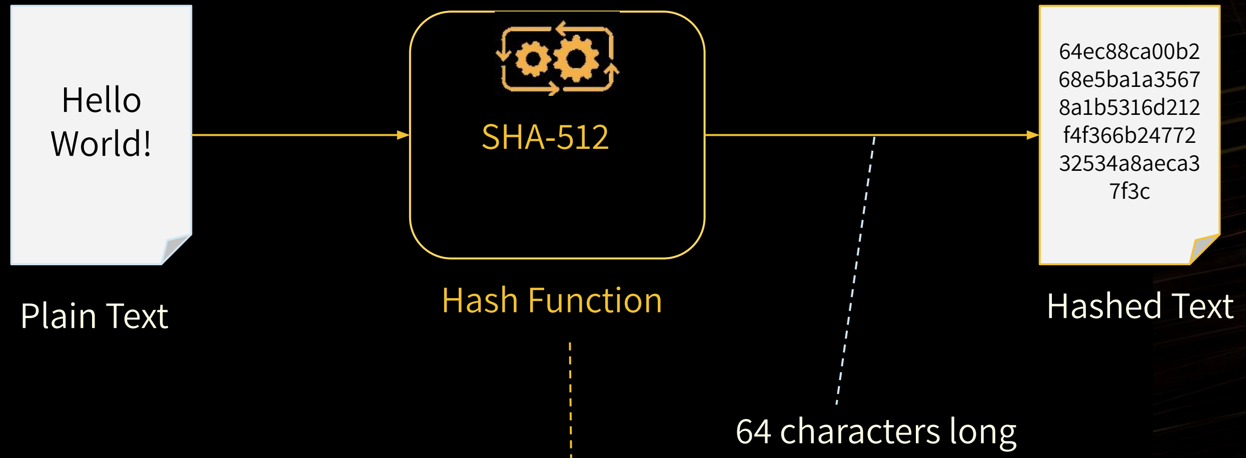
Alice wants to send a secret message to Bob





- RSA (Ron Rivest, Adi Shamir, Leonard Adleman) + key length (128, 192, 256)
- DSA (Data Encryption Standard) - 3DES
- ECC (Elliptical Curve Cryptography) - ECDSA, Ed25519, ECDH

Cryptography 101: Hashing Functions (1 of 2)



- SHA-2 (Secure Hash Algorithm) - SHA-256, SHA-384, SHA-512
- SHA-3

Cryptography 101: Hashing Functions (2 of 2)

The SHA Hash Family: SHA-256, SHA-384, SHA-512, SHA3-256,...

Number of Bits (8 bits = 1 byte)

256/8 = 32
384/8 = 48
512/8 = 64

```
# echo -n "Hello World!" | openssl dgst -sha256
SHA2-256(stdin)= 7f83b1657ff1fc53b92dc18148a1d65dfc2d4b1fa3d677284addd200126d9069
```

```
SHA2-256(stdin)= 7f83b1657ff1fc53b92dc18148a1d65dfc2d4b1fa3d677284addd200126d9069
SHA2-384(stdin)= bfd76c0ebbd006fee583410547c1887b0292be76d582d96c242d2a792723e3fd6fd061f9d5cfd13b8f961358e6adba4a
SHA2-512(stdin)= 861844d6704e8573fec34d967e20bcfef3d424cf48be04e6dc08f2bd58c729743371015ead891cc3cf1c9d34b49264b510751b1ff9e537937bc46b5d6ff4ecc8
```

More hash bits == Higher collision resistance == More secure

Classical

Bit

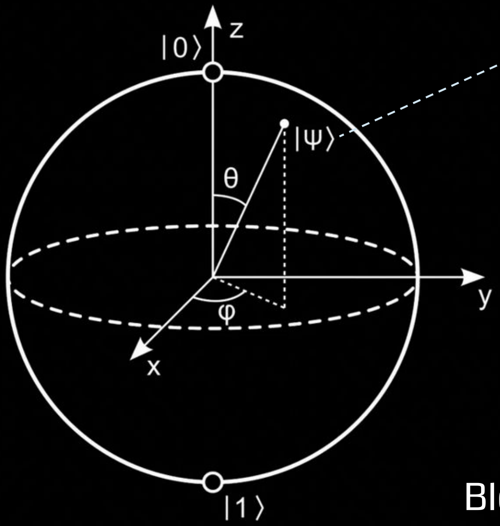
0

or

1

Quantum Computing

Qubit



$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$$

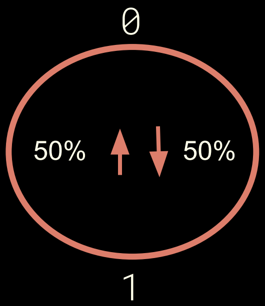
Qubit Quantum State

Bloch Sphere

Quantum Computing

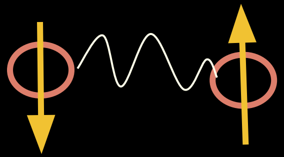
Superposition

Allows **Quantum Bits** or **qubits** to represent both 0 and 1 simultaneously



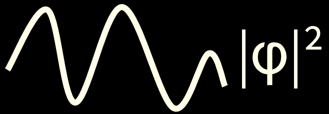
Entanglement

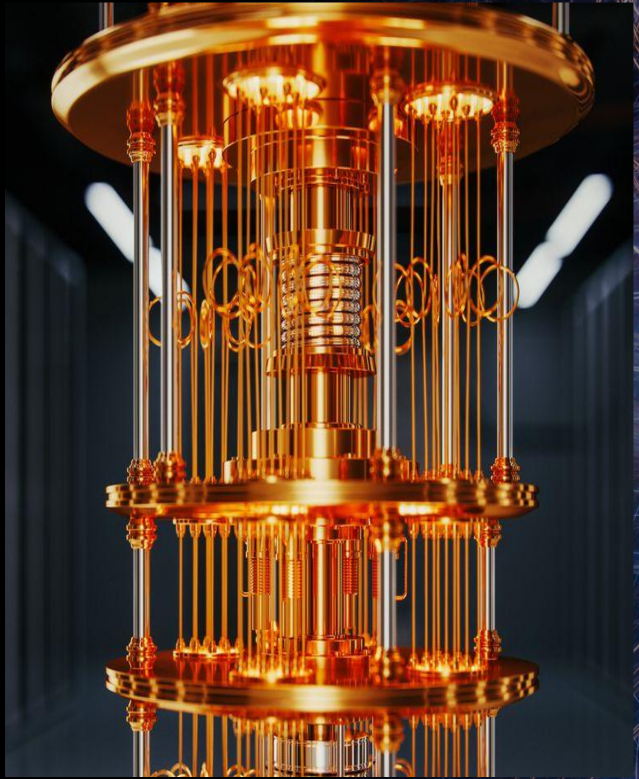
Can perform multiple calculations at once



Interference

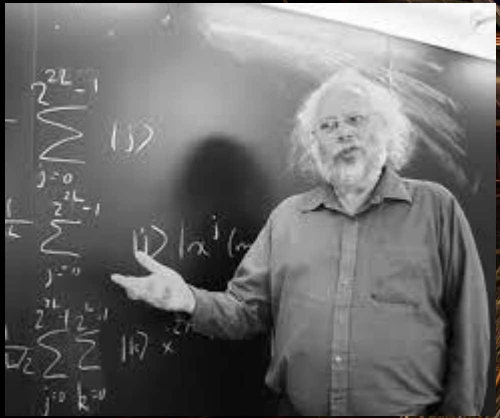
Provides computational speedups by manipulating probabilities

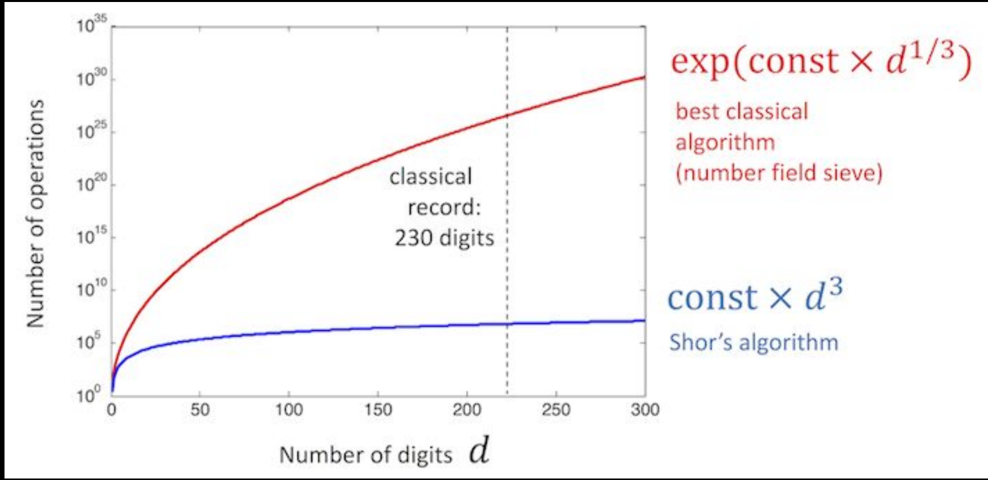




Shor’s Algorithm

- Shor’s Algorithm was developed in 1994 by Peter Shor
- Leverages quantum **superposition** and **entanglement**
- Good at finding prime factors of large integers
- Reduces the time of factoring and discrete logarithm problems
- **Impacts** RSA, DSA, ECC, DIffie-Hellman, etc



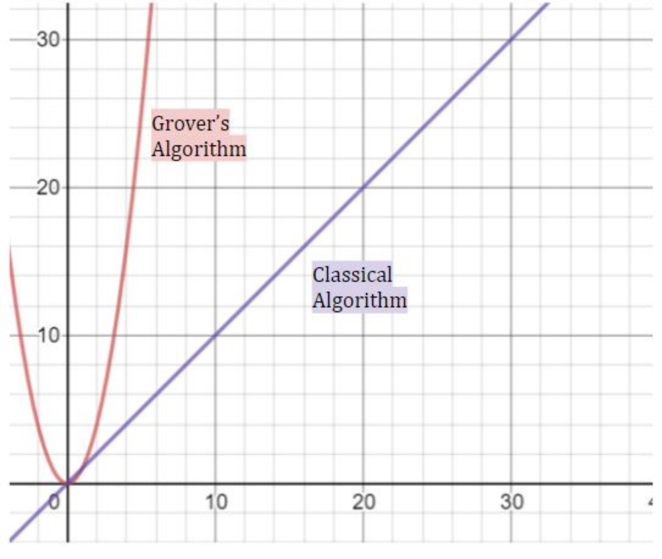


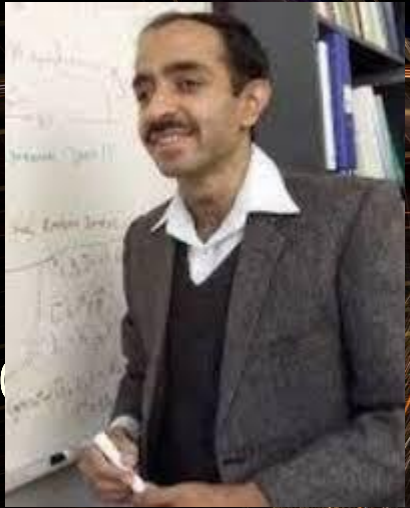
Speedup: $O((\log n)^2) \rightarrow O(2^n)$

The time to break goes from billions of years to weeks or less

Grover’s Algorithm

- Quantum search algorithm developed in 1996
- Leverages quantum superposition and phase interference
- Great a unstructured search problems
- Impacts symmetric key algorithms (AES, 3DES) and hashing functions (





Lov Kumar
Grover

Speedup: $O(n) \rightarrow O(\sqrt{n})$

The time to break a key goes from many billions of years to fewer billions using a brute force attack

NIST Post-Quantum Encryption Standards

M L-KEM

(formerly CRYSTALS-Kyber)

Module Lattice based Key Encapsulation Mechanism (ML-KEM)

For asymmetric encryption

SLH-SSA

(formerly SPINCS+)

Stateless Hash-Based Digital Signature Algorithm (SLH-DSA)

For digital signature

ML-DSA

(formerly CRYSTALS-Dilithium)

Module-Lattice based Digital Signature Algorithm (ML-DSA)

For digital signatures

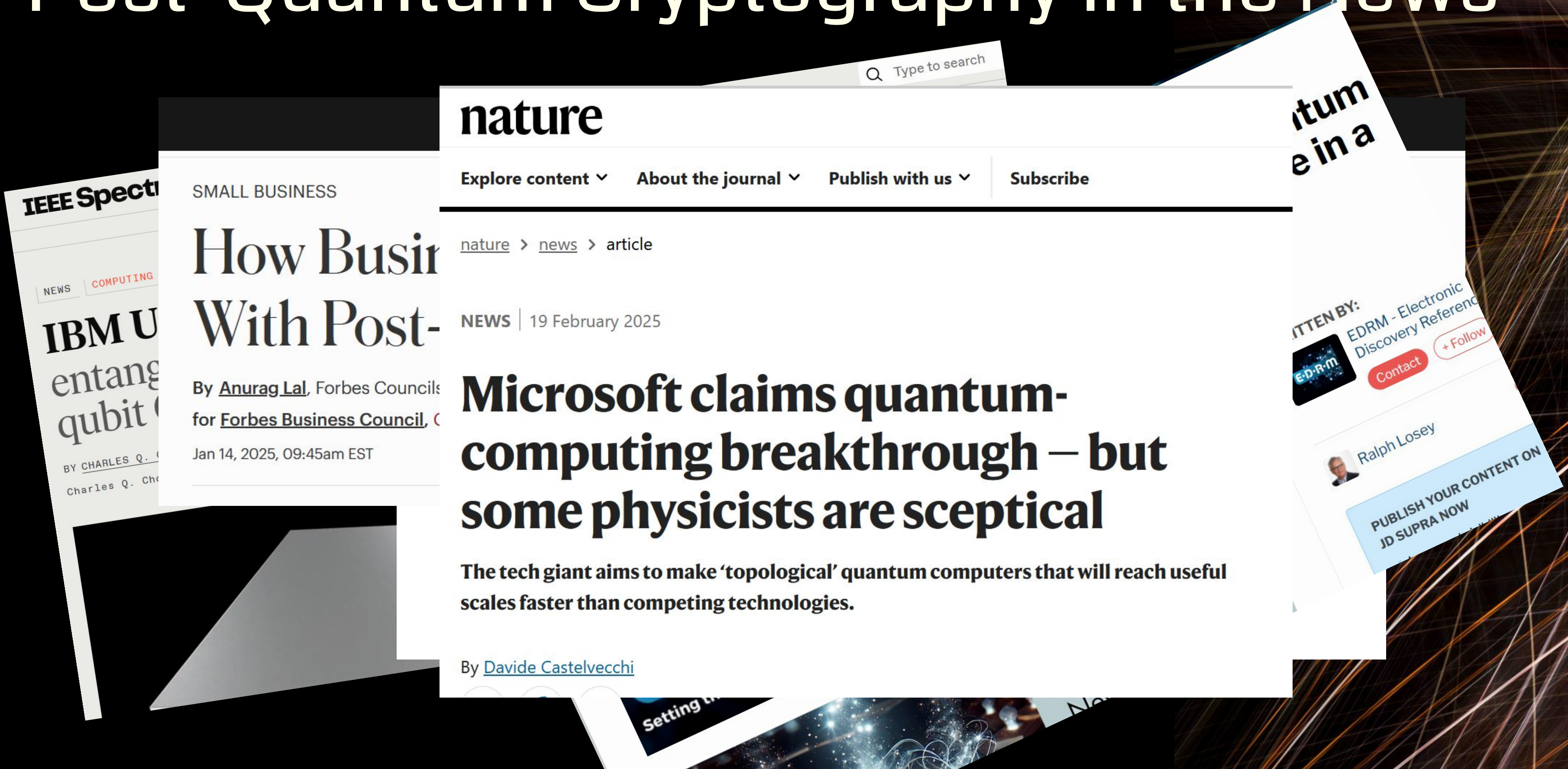
FN-DSA

(formerly FALCON)

Module-Lattice based Digital Signature Algorithm (FN-DSA)

For digital signature

Post-Quantum Cryptography in the News

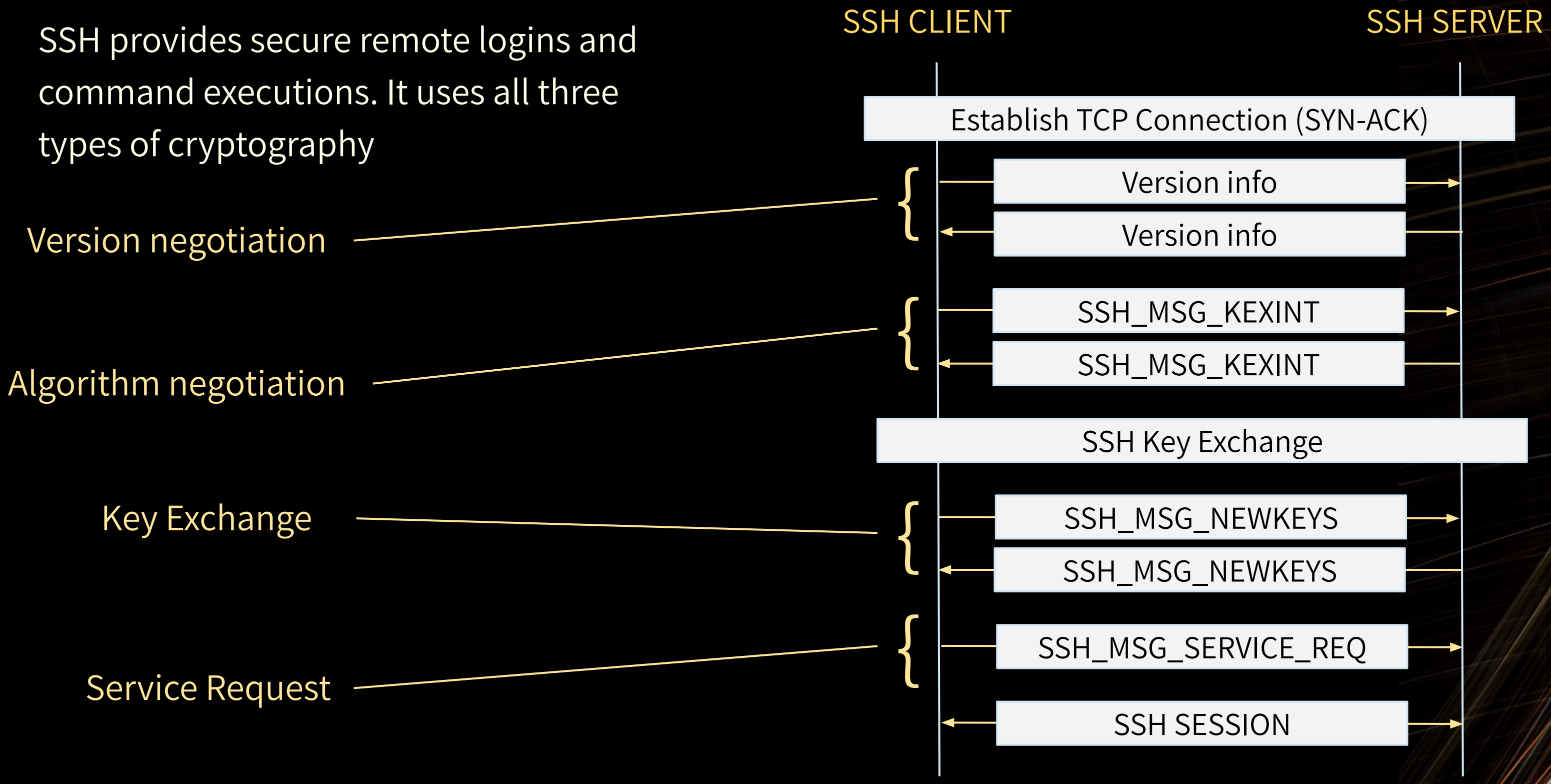


At Risk Cryptographic Algorithms

Algorithm	Type	Purpose	Mitigation
AES-256	Symmetric	Encryption	Larger Key Sizes
SHA-256, SHA-3	Hash	Hash Functions	Larger Key Sizes
RSA	Asymmetric	Signatures/Key Establishment	Not Secure
ECDSA, ECDH	Asymmetric	Signatures/Key Establishment	Not Secure

Secure Shell (SSH) and Cryptography

SSH provides secure remote logins and command executions. It uses all three types of cryptography



HTTPS/TLS and Cryptography

Transport Layer Security (TLS) protocol encrypts data to protect it from authorized access. HTTPS uses TLS to encrypt communication between browser and web server

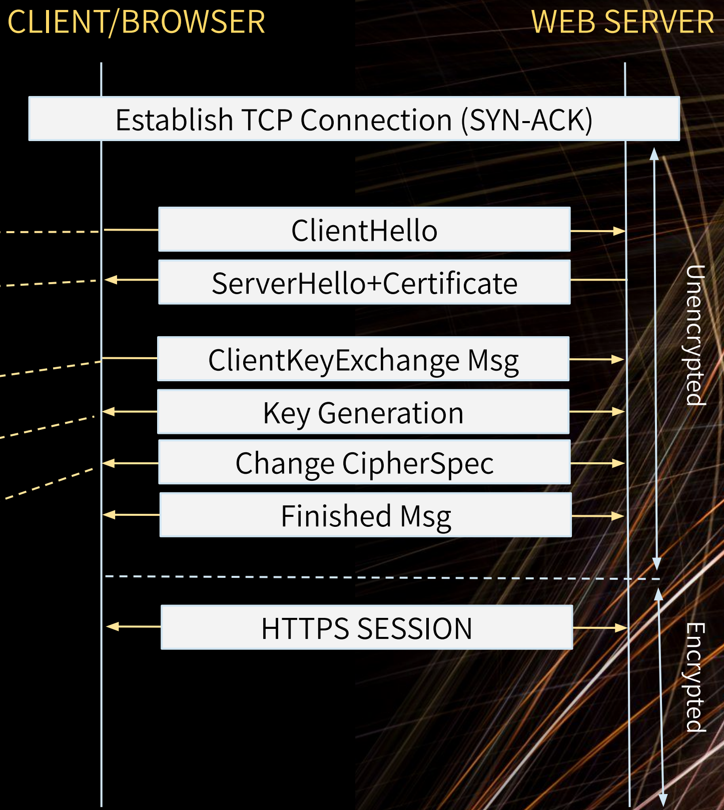
Client Hello
TLS Version, SessionID, List of Cipher Suites

Server Hello
TLS Version, SessionID, Selected Cipher Suites, Server Certificate

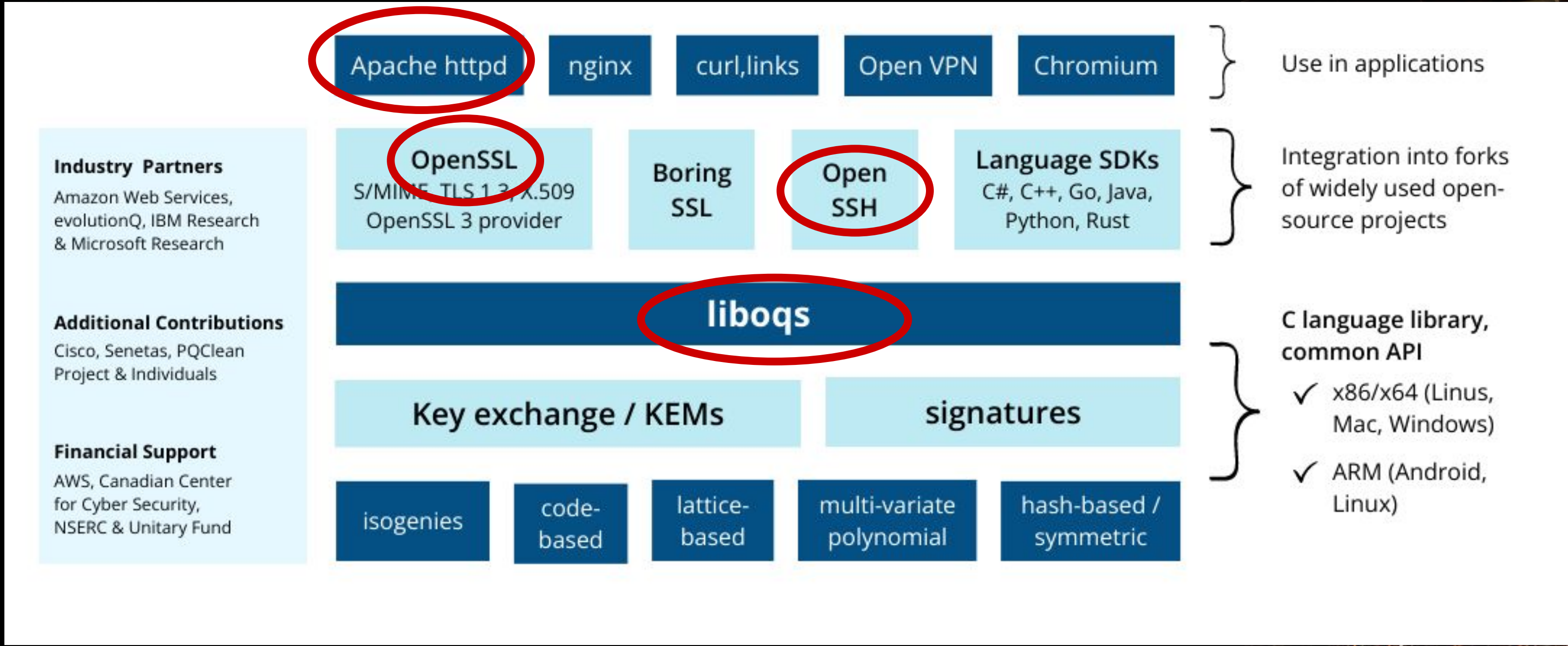
Client Key Exchange
Client generates a shared secret, encrypts it with the public key given in the servers certificate

Key Generation
Client and Server generate a shared key from the key exchange

Change Cipher Spec
Start Using the new keys for encryption



Open Quantum Safe (OQS) Project



Enabling PQC in OpenSSL

1. Build and install liboqs
2. Build and install oqs provider
3. Configure OpenSSL w/oqs provider
4. Verify OpenSSL configuration

Our Environment

Virtual Machine: Redhat 9.5 minimal install

Dependencies:

- Git, GCC development tools
- OpenSSL 3.2.2 + devel packages

Step 1: Build and install liboqs

// Create the source and build directory

mkdir -p ~src/liboqs && cd ~/src/liboqs

// Download liboqs source from git repo

git clone https://github.com/open-quantum-safe/liboqs.git

cd liboqs

// Set environment with path the liboqs and openssl

mkdir build && cd build

cmake   -DBUILD_SHARED_LIBS=ON   -DOQS_USE_OPENSSL=OFF \

 -DCMAKE_BUILD_TYPE=Release   -DOQS_BUILD_ONLY_LIB=ON-DOQS_DIST_BUILD=ON \

..

// build and install to /usr/local/lib64

make -j $(nproc)

make -j install

Step 2: Build and install oqs-provider

// Create the source and build directory

mkdir -p ~src/oqs-provider && cd ~/src/oqs-provider

// Download oqs-provider source from git repo

git clone https://github.com/open-quantum-safe/oqs-provider.git

cd oqs-provider

// Set environment with path the liboqs and openssl

export liboqs_DIR=/usr/local/lib64

export OPENSSL_INSTALL=/usr

// Build and install to /usr/local/lib64

scripts/fullbuild.sh

cmake -–install _build

Step 3: Configure OpenSSL w/oqs-provider

// Using vi, add the oqs-provider configuration to ssl.conf

vi /etc/ssl/openssl.cnf

```
# PQC oqs-provider
[provider_sect]
default = default_sect
oqsprovider = oqsprovider_sect

[default_sect]
activate = 1

[oqsprovider_sect]
activate = 1
```

Define the osq provider

Activate the osq provider

Step 4a: Verify oqs-provider configuration

// List the PQC provider

```
# openssl list -providers

 Providers:
  default
        name: OpenSSL Default Provider
        version: 3.2.2
        status: active
  oqsprovider
        name: OpenSSL OQS Provider
        version: 0.8.1-dev
        status: active
```

// List the ML KEM algorithms

```
#  openssl list -kem-algorithms | grep kem

  mlkem512 @ oqsprovider
  p256_mlkem512 @ oqsprovider
  x25519_mlkem512 @ oqsprovider
  mlkem768 @ oqsprovider
  p384_mlkem768 @ oqsprovider
  x448_mlkem768 @ oqsprovider
  mlkem1024 @ oqsprovider
  p521_mlkem1024 @ oqsprovider
```

Step 4b: Verify oqs-provider configuration

// List the KEX algorithms

```
#  openssl list -key-exchange-algorithms
[...snip…]
  p256_mldsa44 @ oqsprovider
  rsa3072_mldsa44 @ oqsprovider
  mldsa44_pss2048 @ oqsprovider

  falcon512 @ oqsprovider
  p256_falcon512 @ oqsprovider
  rsa3072_falcon512 @ oqsprovider
  falconpadded512 @ oqsprovider

  rsa3072_sphincssha2128ssimple @ oqsprovider
  sphincssha2192fsimple @ oqsprovider
```

Implementing PQC in Apache

1. Install Apache web server
2. Enable X25519MLKEM768
3. Test your HTTPS in Apache

Our Environment

Virtual Machine: Redhat 9.5 minimal install

Dependencies:

- OpenSSL w/oqs-provider
- Apache/mod_ssl

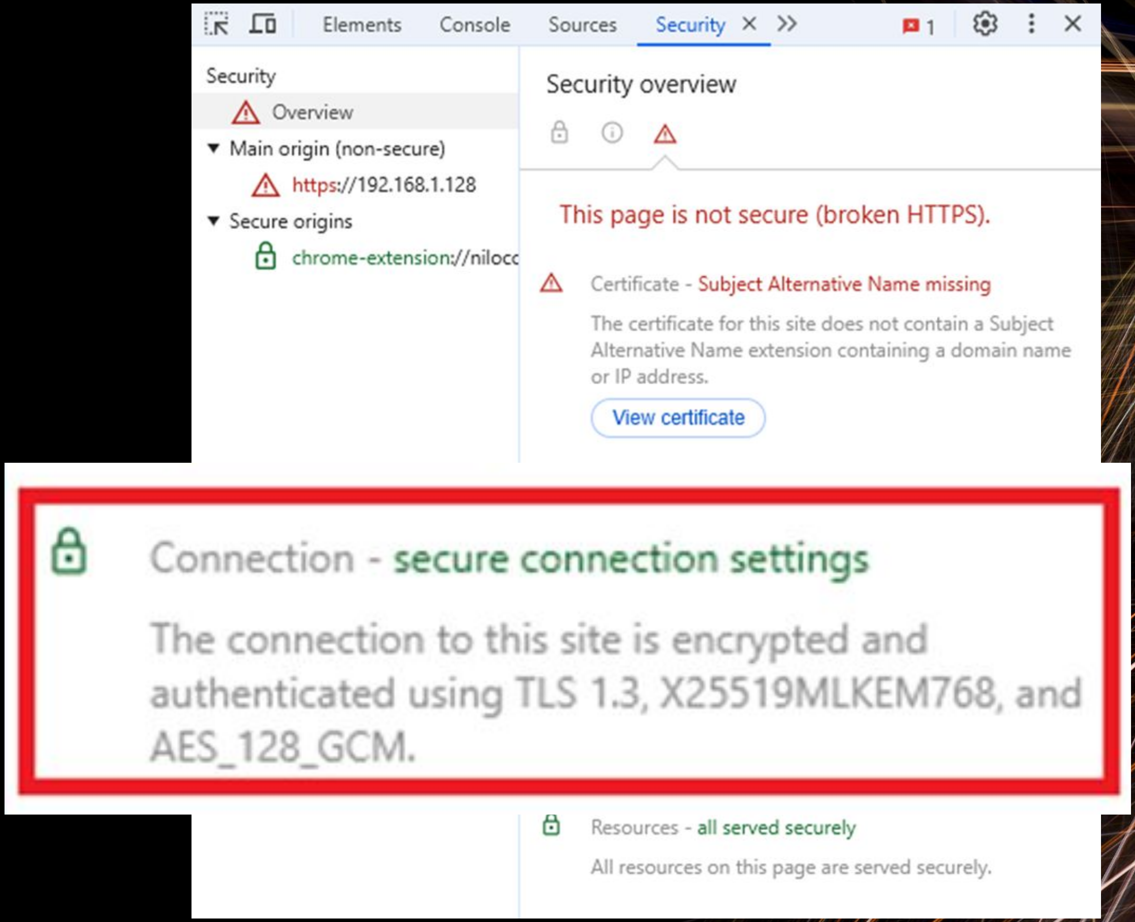
```
// Step 1: Install Apache with mod_ssl
# dnf install httpd mod_ssl
```

Step 2: Enable PQC algorithm

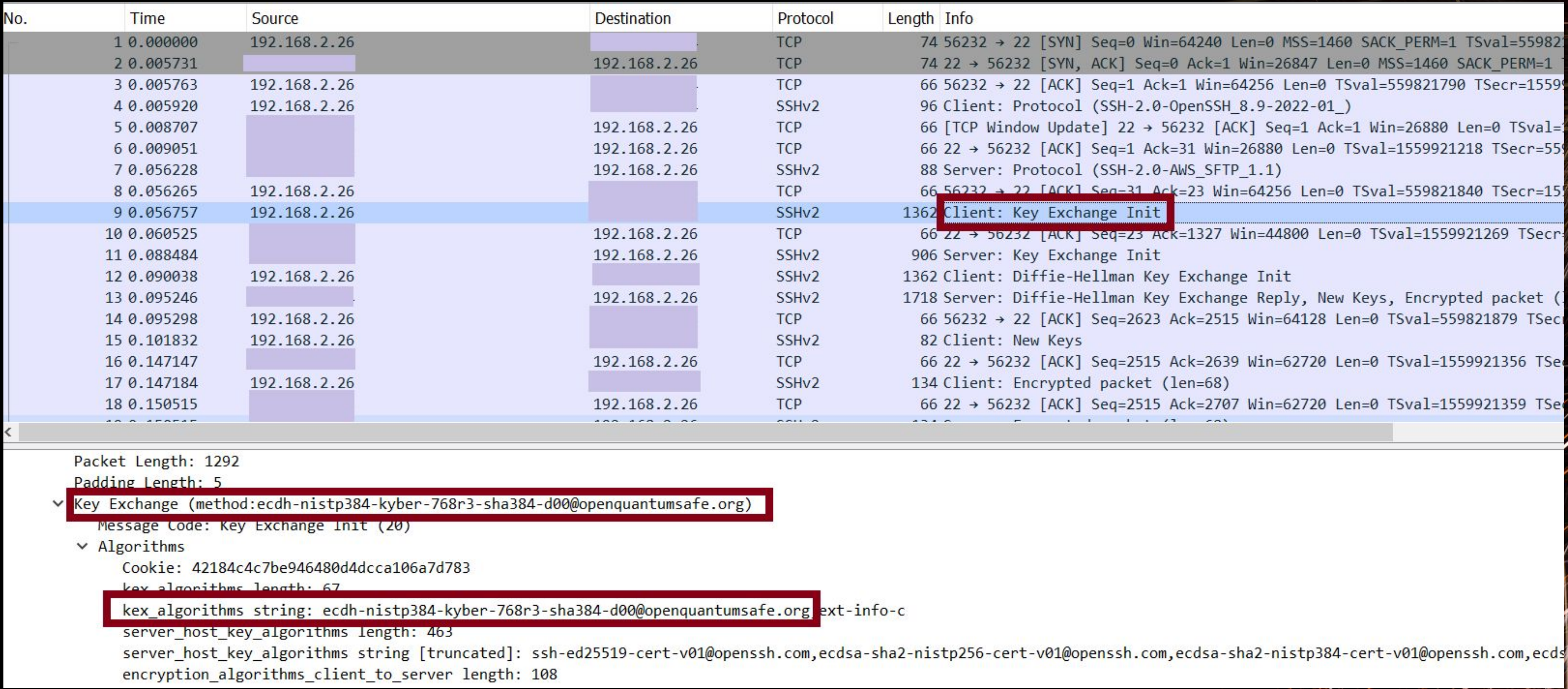
```
// Configure ssl configuration with Key Encapsulation Mechanism algorithm
# vi /etc/apache/conf.d/ssl.conf

# Configure key exchange and key encapsulation mechanisms
SSLOpenSSLConfCmd Curves X25519MLKEM768:X448:X25519:prime256v1
```

```
// Restart apache web server
# systemctl restart httpd
```

Step 3: Test Apache PQC Connection



Wireshark Screenshot of SSH Session



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.2.26		TCP	74	56232 → 22 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=55982
2	0.005731		192.168.2.26	TCP	74	22 → 56232 [SYN, ACK] Seq=0 Ack=1 Win=26847 Len=0 MSS=1460 SACK_PERM=1
3	0.005763	192.168.2.26		TCP	66	56232 → 22 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=559821790 TSecr=1559
4	0.005920	192.168.2.26		SSHv2	96	Client: Protocol (SSH-2.0-OpenSSH_8.9-2022-01_)
5	0.008707		192.168.2.26	TCP	66	[TCP Window Update] 22 → 56232 [ACK] Seq=1 Ack=1 Win=26880 Len=0 TSval=
6	0.009051		192.168.2.26	TCP	66	22 → 56232 [ACK] Seq=1 Ack=31 Win=26880 Len=0 TSval=1559921218 TSecr=55
7	0.056228		192.168.2.26	SSHv2	88	Server: Protocol (SSH-2.0-AWS_SFTP_1.1)
8	0.056265	192.168.2.26		TCP	66	56232 → 22 [ACK] Seq=31 Ack=23 Win=64256 Len=0 TSval=559821840 TSecr=15
9	0.056757	192.168.2.26		SSHv2	1362	Client: Key Exchange Init
10	0.060525		192.168.2.26	TCP	66	22 → 56232 [ACK] Seq=23 Ack=1327 Win=44800 Len=0 TSval=1559921269 TSecr
11	0.088484		192.168.2.26	SSHv2	906	Server: Key Exchange Init
12	0.090038	192.168.2.26		SSHv2	1362	Client: Diffie-Hellman Key Exchange Init
13	0.095246		192.168.2.26	SSHv2	1718	Server: Diffie-Hellman Key Exchange Reply, New Keys, Encrypted packet (
14	0.095298	192.168.2.26		TCP	66	56232 → 22 [ACK] Seq=2623 Ack=2515 Win=64128 Len=0 TSval=559821879 TSec
15	0.101832	192.168.2.26		SSHv2	82	Client: New Keys
16	0.147147		192.168.2.26	TCP	66	22 → 56232 [ACK] Seq=2515 Ack=2639 Win=62720 Len=0 TSval=1559921356 TSe
17	0.147184	192.168.2.26		SSHv2	134	Client: Encrypted packet (len=68)
18	0.150515		192.168.2.26	TCP	66	22 → 56232 [ACK] Seq=2515 Ack=2707 Win=62720 Len=0 TSval=1559921359 TSe

```
Packet Length: 1292
Padding Length: 5
Key Exchange (method:ecdh-nistp384-kyber-768r3-sha384-d00@openquantumsafe.org)
  Message Code: Key Exchange Init (20)
  Algorithms
    Cookie: 42184c4c7be946480d4dcca106a7d783
    kex_algorithms length: 67
    kex_algorithms string: ecdh-nistp384-kyber-768r3-sha384-d00@openquantumsafe.org,ext-info-c
    server_host_key_algorithms length: 463
    server_host_key_algorithms string [truncated]: ssh-ed25519-cert-v01@openssh.com,ecdsa-sha2-nistp256-cert-v01@openssh.com,ecdsa-sha2-nistp384-cert-v01@openssh.com,ecds
    encryption_algorithms_client_to_server length: 108
```

Other Vulnerable Services

Public Key Infrastructure

Certificate Authorities (CA) configured with RSA/ECC keys

Blockchain and Crypto Currencies

Bitcoin - ECDSA / SHA-256
Ethereum - ECDSA / Keccak (SHA-3)
Litecoin - ECDSA / SCRYPT

PGP/GPG Encryption

Secure Email

Who's Doing PQC?

- Mozilla using X25519+Kyber
- OpenSSH 9.0
- Google Chrome
- Microsoft Edge
- Cloudflare front end sites using X25529+Kyber
- Signal using Post-Quantum Extended DIffie-Hellman (PQXDH)
- Amazon
- Cisco
- Proton Mail

Next Steps?

- Quantum threats are coming … but Don’t Panic, there’s still time
- Start testing PQC implementations and configurations
- Practice Crypto-Agility
- Start testing PQC today!

Encrypt …
Like it's 2030

Thank You!

Do you have any questions?

Atom - aramirez415@ucmerced.edu / LinkedIn

Marie - mariecurieramirez@berkeley.edu / LinkedIn