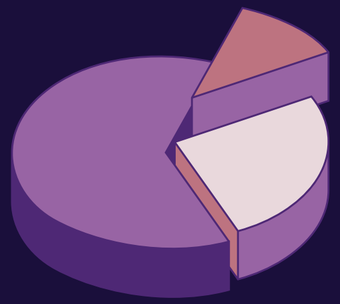


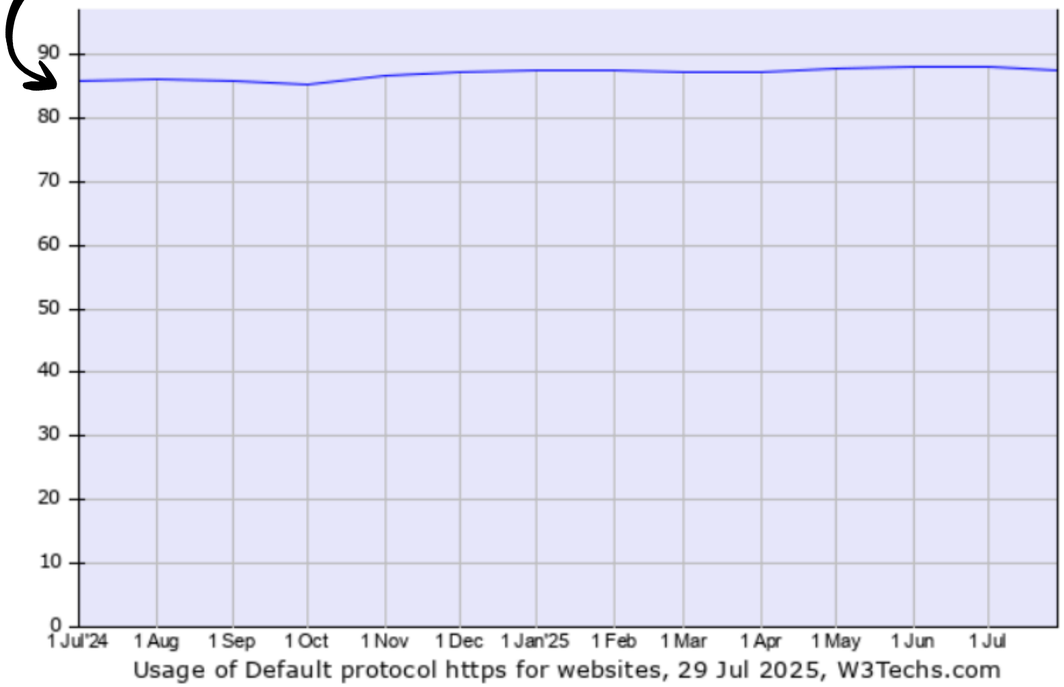
Adoption Trends & Cryptographic Algorithms in Internet Communication Protocols??

(TLS variants, QUIC, IPSec, OAuth, SSH)

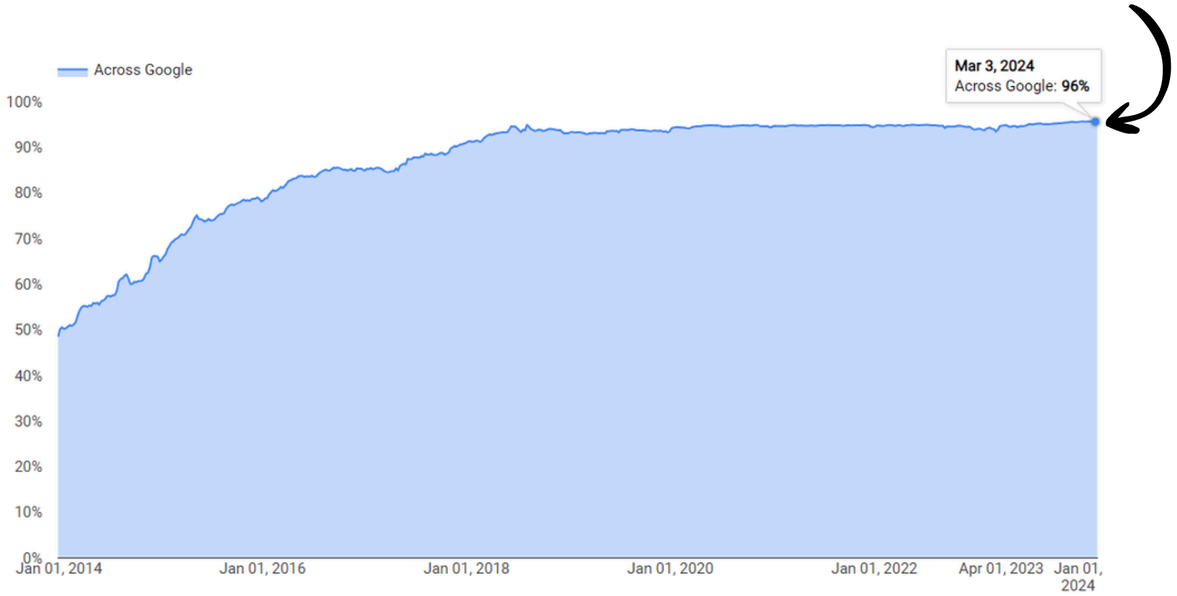
Adoption Stats and Algorithm: TLS (and its variants)

- **ECDHE key exchange** with **P-256/P-384 curves**, **ECDSA/RSA authentication**
- Cipher suites: **TLS_AES_256_GCM_SHA384** with **ECDHE-P256 key agreement**

87.6% of the Websites Use a Valid SSL Certificate



96% of Chrome browsing time occurs over HTTPS connections



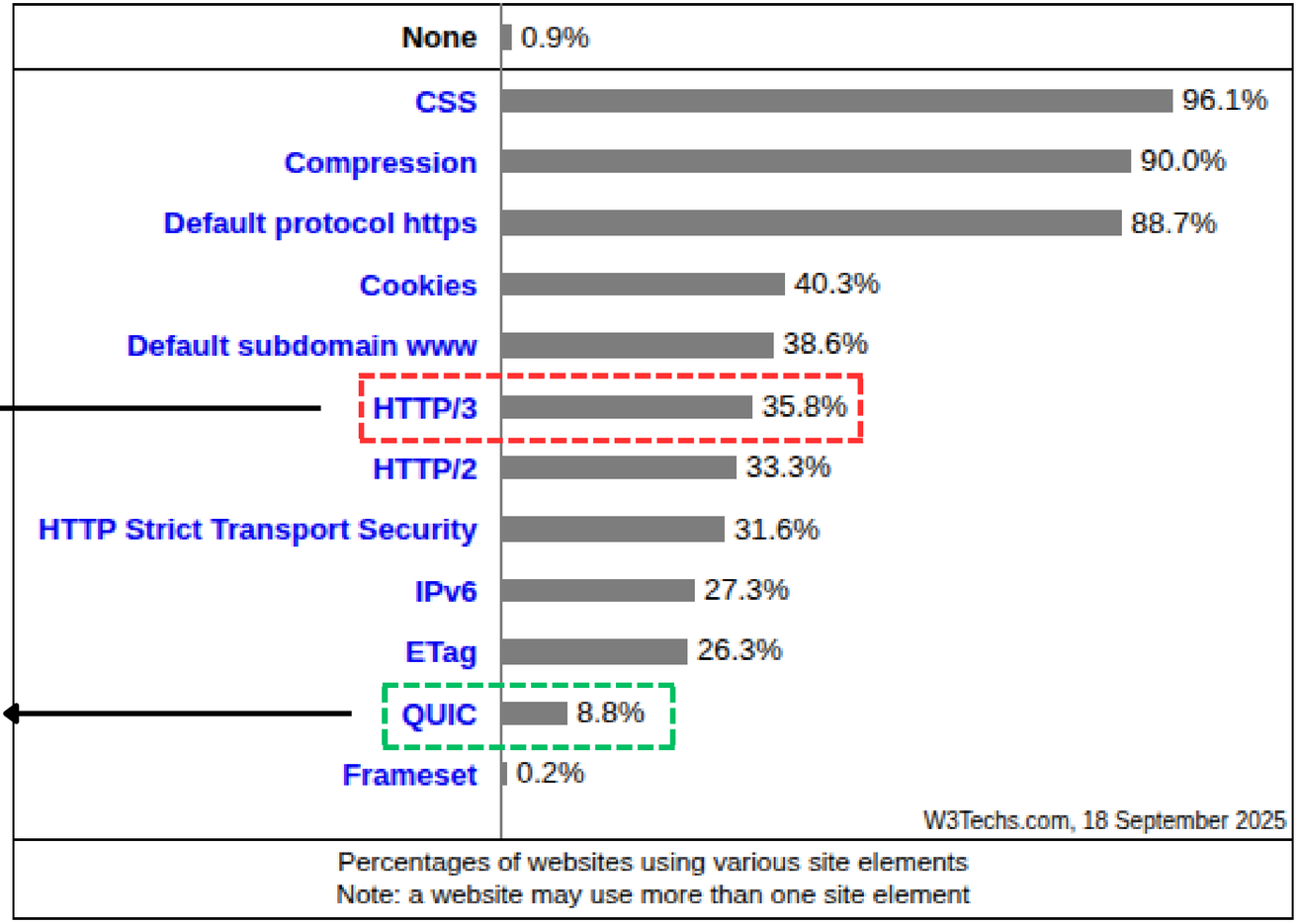
Source: *https://sslinsights.com/ssl-certificates-statistics/*

Adoption Stats and Algorithm: QUIC

- **Mandatory TLS 1.3** integration for all QUIC implementations
- Inherits **TLS 1.3's ECDHE key exchange** and **ECDSA/RSA signatures**

About 35.8 % of websites advertise HTTP/3 support (via Alt-Svc headers or DNS)

About 8.8 % of websites include QUIC in their tech stack

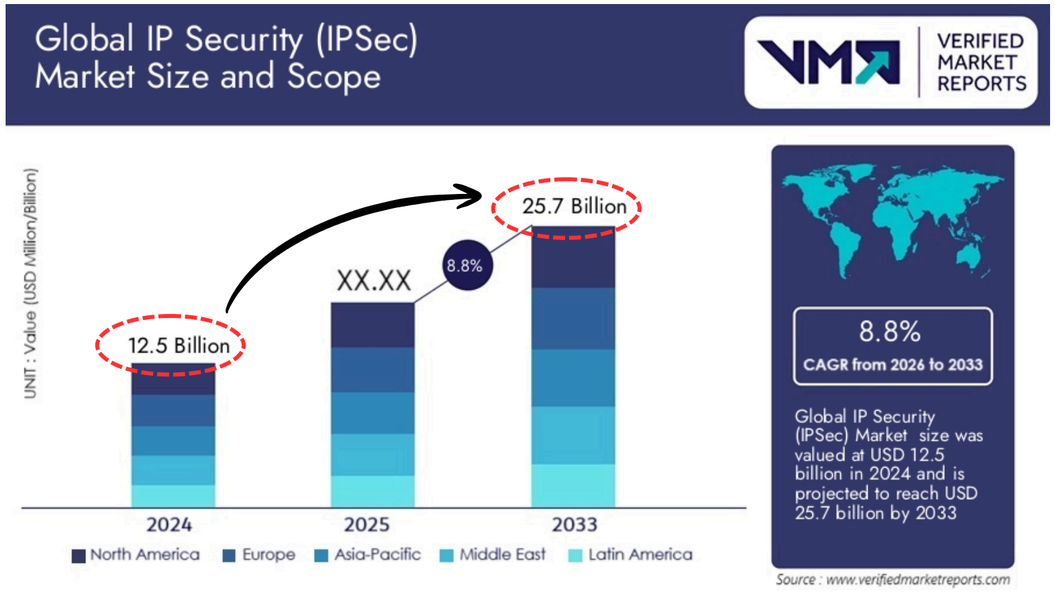


Source: *https://w3techs.com/technologies/overview/site_element*

Adoption Stats and Algorithm: IPSec

- A significant portion of Internet users **(~ 23-25 %) worldwide employ VPN services** for privacy or security as of 2024
- IKEv2 with **ECDHE group 19 (P-256) key exchange** - most common
- Primarily **AES-GCM** with **HMAC-SHA-256 authentication**

Top IP Security (IPSec) Market Companies

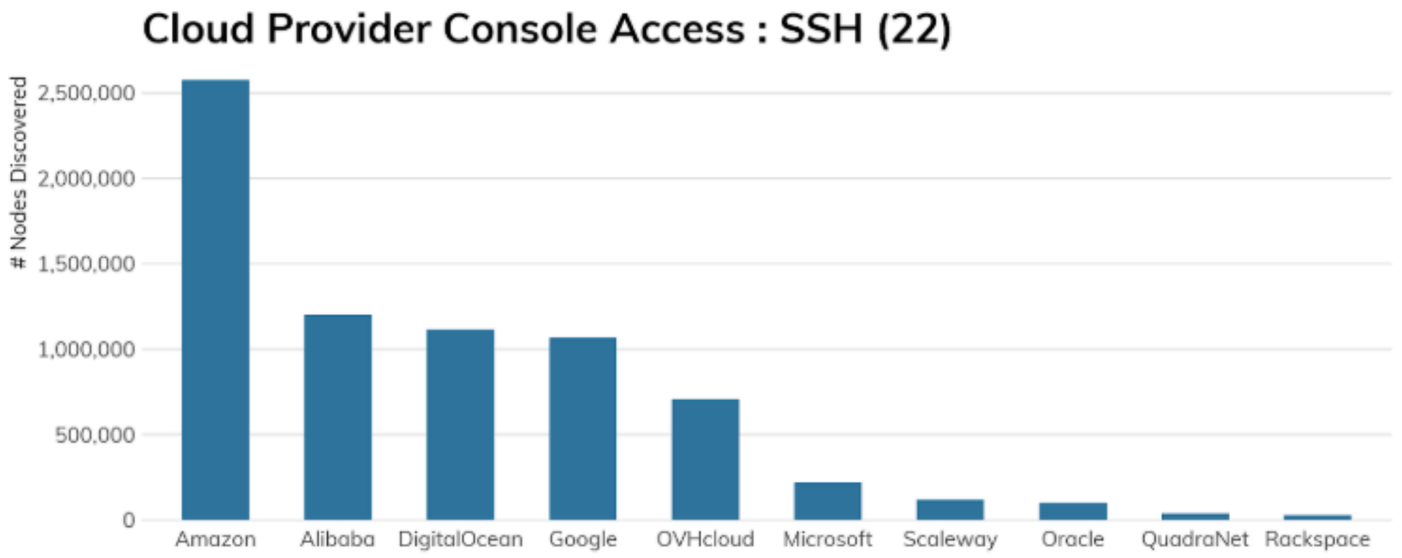


Expected growth of upto 25.7B by 2033, about 8.8% CAGR from 2024

Source: *https://www.verifiedmarketreports.com/product/ip-security-ipsec-market/*

Adoption Stats and Algorithm: SSH

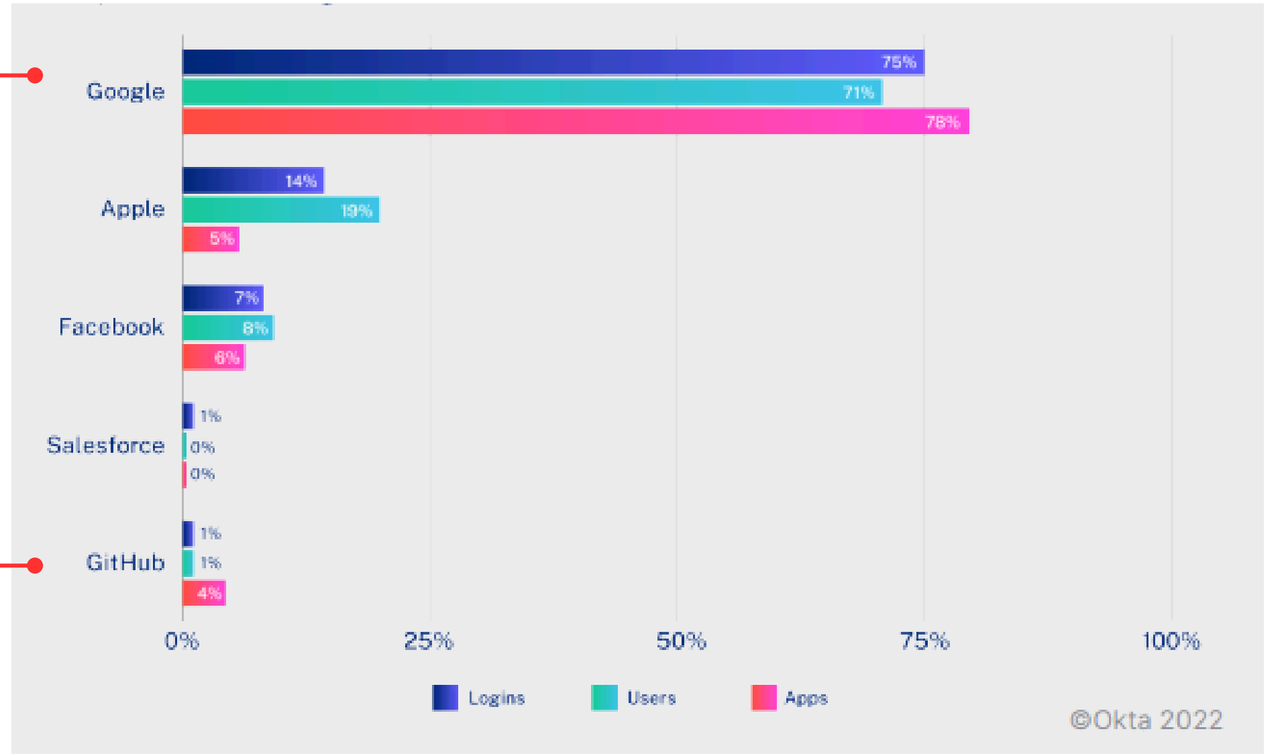
- Key exchange algorithms: **curve25519-sha256** (most common), **ecdh-sha2-nistp256**
- Host key types: **ssh-ed25519, ecdsa-sha2-nistp256, rsa-sha2-512**
- Entire session security depends on **ECDH key exchange**



Source: *https://www.rapid7.com/blog/post/2020/08/28/nicer-protocol-deep-dive-secure-shell-ssh/*

Adoption Stats and Algorithm: OAuth

Most used social login methods among companies using Auth0



Source: *https://assets.ctfassets.net/2ntc334xpx65/77U9sLFO7rD7t9zdI6Q1SV/a8e2054b5affc0280769516eee70b0ea/Social-Login-Report.pdf*

- **RSA/ECDSA signatures for JWT validation,** relies on TLS for transport security
- Token integrity depends entirely on **classical signature algorithms**

Why These Numbers Matter

Dependence on Classical Cryptography!!

NGKORE

Domain	Technology	References	Classical Algorithm	Quantum Risk Type
Transport Network	TLS 1.2/1.3 (HTTPS, QUIC, SMTPS/IMAPS, LDAPs)	[12], [22]	RSA, ECDHE, ECDSA	HTDL (ECDHE); cert/signature forgery (RSA/ECDSA)
	IPsec / IKEv2	[13]	RSA, (EC)DH, ECDSA	HTDL + authentication forgery
	SSH	[24], [26]	RSA, ECDSA, Ed25519, X25519	HTDL (KEX); key/signature forgery
	WireGuard	[27]	X25519	HTDL (session decryption)
	Wi-Fi Enterprise (EAP-TLS)	[28]	RSA/ECDSA (X.509), ECDHE	HTDL; cert forgery
	WPA3-SAE	[29]	Finite-field/ECC PAKE	DL break → session compromise
	Bluetooth LE Secure Connections	[30]	ECDH P-256	HTDL; impersonation
	Tor (identities + TLS)	[31]	Ed25519; RSA/ECDSA (TLS)	Identity forgery; HTDL
	Signal (X3DH, Double Ratchet, PQXDH)	[32], [20]	X25519, Ed25519; hybrid ML-KEM	Legacy HTDL + signature forgery; PQ-resistant for session keys (PQXDH)
Web Mail	Web PKI (X.509)	[33]	RSA, ECDSA	Certificate forgery
	S/MIME (CMS)	[34], [35], [36]	RSA, ECDSA, ECDH	SignedData forgery; HTDL decryption
	OpenPGP / GPG	[37]	RSA, ECC, ElGamal	Signature forgery; HTDL if EC-DH used
	DKIM	[41], [42]	RSA; Ed25519/ECDSA	Domain signature forgery
Identity SSO	JWT/JWS (OAuth2, OIDC)	[43], [44], [45]	RS256/PS256/ES256	Token forgery
	SAML 2.0 (XMLDSIG)	[46]	RSA, ECDSA	Assertion forgery
	WebAuthn / FIDO2	[47], [48]	ES256, EdDSA	Credential forgery
Infra	DNSSEC	[49], [50]	RSA, ECDSA, EdDSA	Zone/RRset forgery
	RPKI / BGPsec	[51], [52]	RSA, ECDSA	ROA/path forgery
	Certificate Transparency	[53]	RSA, ECDSA	Log/STH forgery
Software Supply Chain	Code signing (OS, apps, APKs)	[54], [55], [57]	RSA, ECDSA (X.509)	Binary/firmware forgery
	Package managers (APT, RPM, etc.)	[58], [60]	OpenPGP/GPG	Repository/package forgery
	Secure Boot (UEFI)	[62]	RSA-2048; ECDSA-384	Bootloader/firmware forgery
	Sigstore / Cosign	[67]	X.509/OIDC (RSA/ECC)	Container/artifact forgery
	Git (commits, tags)	[68]	OpenPGP; X.509; SSH	Commit/tag forgery
Docs Records	PAdES, CAdES, XAdES	[69], [70], [71], [72]	RSA, ECDSA (CMS/X.509)	Document/signature forgery
	EMV (chip, POS, NFC)	[75], [77], [78]	RSA-2048, ECC P-256	Offline Tx forgery
	ePassports / eIDs	[81], [82]	RSA, ECDSA	CSCA/DS forgery
	Blockchain (Bitcoin, Ethereum, Solana)	[84], [87], [89]	ECDSA secp256k1; Ed25519	Tx/signature forgery; HTDL

Source: *https://arxiv.org/pdf/2509.24623*

Progress Towards Quantum-Resilient Security: Advancement in PQC

Foundation for PQC

NIST Releases First 3 Finalized Post-Quantum Encryption Standards

August 13, 2024

While there have been no substantive changes made to the standards since the draft versions, NIST has changed the algorithms' names to specify the versions that appear in the three finalized standards, which are:

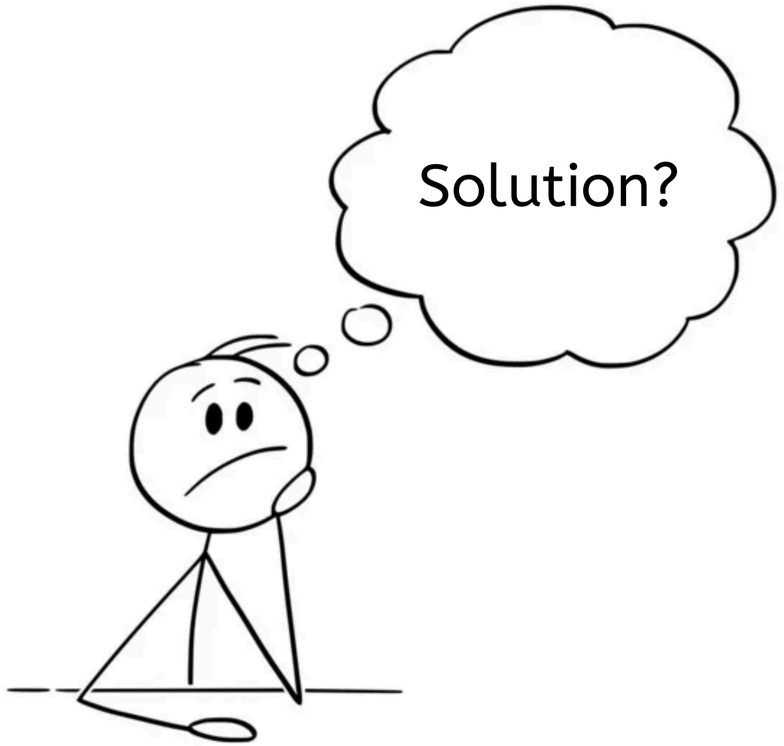
- **Federal Information Processing Standard (FIPS) 203**, intended as the primary standard for general encryption. Among its advantages are comparatively small encryption keys that two parties can exchange easily, as well as its speed of operation. The standard is based on the CRYSTALS-Kyber algorithm, which has been renamed ML-KEM, short for Module-Lattice-Based Key-Encapsulation Mechanism.
- **FIPS 204**, intended as the primary standard for protecting digital signatures. The standard uses the CRYSTALS-Dilithium algorithm, which has been renamed ML-DSA, short for Module-Lattice-Based Digital Signature Algorithm.
- **FIPS 205**, also designed for digital signatures. The standard employs the Sphincs+ algorithm, which has been renamed SLH-DSA, short for Stateless Hash-Based Digital Signature Algorithm. The standard is based on a different math approach than ML-DSA, and it is intended as a backup method in case ML-DSA proves vulnerable.

- **Major security bodies—CISA, NSA, NCSC UK, DoD, and ENISA**—immediately endorsed these standards and **published migration guidance**
- **Federal agencies mandated inventory and migration plans** via OMB M-23-02 and NSM-10, with **2035 as the target deadline**
- **EU and UK national cybersecurity centers set PQC migration timelines through 2035**, requiring phased deployments and annual progress reporting
- **Cloud providers** (AWS, Azure, Google Cloud) **published PQC policy pages enforcing hybrid key exchanges** and **PQC-compatible certificates** for all new workloads

Source: *https://www.nist.gov/news-events/news/2024/08/nist-releases-first-3-finalized-post-quantum-encryption-standards*

Where’s the Problem?

- **PQ versions of major protocols** —TLS, SSH, IPsec, OAuth, and DTLS— **are not fully standardized**
- **Waiting is risky. Delaying migration invites “harvest-now, decrypt-later” attacks**, where adversaries capture encrypted data today for decryption once quantum computers become practical



October 9, 2025

Protocol Migration in Practice: From Classical to Post-Quantum Cryptography with OpenSSL

OpenSSL Conference Prague 2025

Who Are We?

Shankar Malik

- Governing Board Committer Representative ***@LF Networking (LFN)***
- TSC Member ***@5G 5G SBP*** *&* ***@L3AF Project***
- Chief Architect ***@NgKore Community (Co-founder)***

Shubham Kumar

- Mentor **@LF Decentralized Trust (LFDT)**
- NgKore's Representative **@PKI Consortium**
- SW Engineer ***@NgKore Community (Co-founder)***

Agenda

- Adoption Trends & Cryptographic Algorithms in Internet Protocols
- Risks in Security Protocols – Dependence on Classical Cryptography
- Progress Towards Quantum-Resilient Security (Foundation for PQC)
- Identifying the Problem: The Gap
- Session Overview & About Us
- Project Overview – pqproto
- TLS
  - mTLS
  - DTLS
- QUIC
- IPSec
- SSH & OAuth (WIP)
- Challenges in PQC Migration
- Migration Strategy & Best Practices
- Our Implementations & Real-World Applications

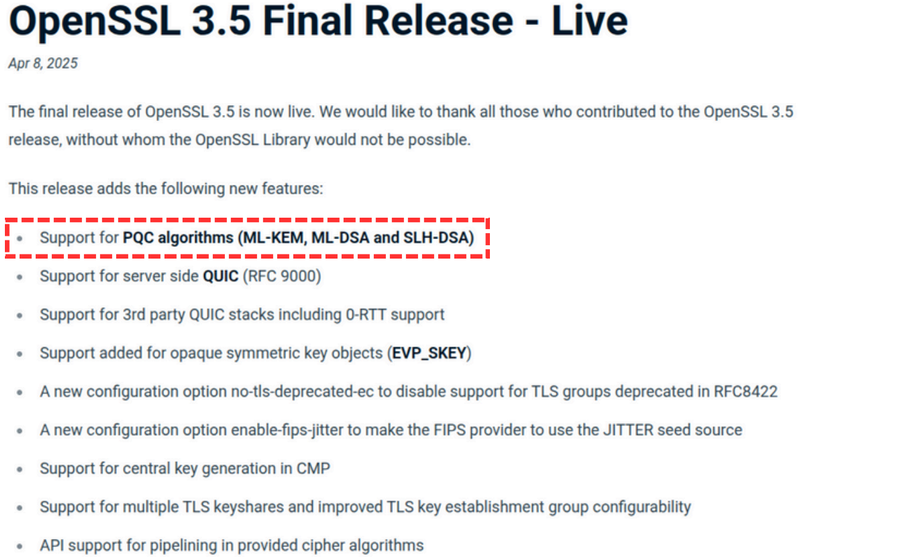
Project Overview – pqproto

A simplified prototype showcasing classical and post-quantum secure client–server communication.

- **Purpose:** Demonstrates how to migrate classical cryptographic protocols to post-quantum (PQ) alternatives.
- **Protocols Covered:** TLS, mTLS, DTLS, QUIC, IPsec, OAuth, and SSH (more in the future)
- **Implementation Style:** Command-line (CLI)–based client–server setup for easy testing and automation
- **Core Libraries & Tools:**
  - OpenSSL with PQC support (v3.5.0 & above)
  - liboqs and oqs-provider for hybrid/PQ key exchange and signatures
  - strongSwan for IPsec integration
  - Circl and other PQC libraries for algorithm diversity
- Repository: *https://github.com/ngkore/pqproto*

TLS (Transport Layer Security)

- A cryptographic protocol that encrypts data to secure communications over the internet
- Core Guarantees: Confidentiality, Integrity, Authentication
- Operates over TCP at the transport layer
- Negotiates the highest supported version (typically TLS 1.3 or TLS 1.2)
- OpenSSL 3.5 TLS Guide: *https://docs.openssl.org/3.5/man7/ossl-guide-tls-introduction/*
- Simple Echo Client/Server Demo: *https://github.com/openssl/openssl/tree/master/demos/sslecho*
- pqproto Classical and PQ-Support TLS1.3 Client/Server Setup - *https://github.com/ngkore/pqproto/tree/main/tls*



OpenSSL 3.5 Final Release - Live

Apr 8, 2025

The final release of OpenSSL 3.5 is now live. We would like to thank all those who contributed to the OpenSSL 3.5 release, without whom the OpenSSL Library would not be possible.

This release adds the following new features:

- Support for **PQC algorithms (ML-KEM, ML-DSA and SLH-DSA)**
- Support for server side **QUIC** (RFC 9000)
- Support for 3rd party QUIC stacks including 0-RTT support
- Support added for opaque symmetric key objects (**EVP_SKEY**)
- A new configuration option no-tls-deprecated-ec to disable support for TLS groups deprecated in RFC8422
- A new configuration option enable-fips-jitter to make the FIPS provider to use the JITTER seed source
- Support for central key generation in CMP
- Support for multiple TLS keyshares and improved TLS key establishment group configurability
- API support for pipelining in provided cipher algorithms

Source: *https://openssl-corporation.org/post/2025-04-08-openssl-35-final-release/*

TLS: Algorithms and Cipher Suites

NGKORE

Cipher Suites

```
ubuntu@strongswan:~/pqproto/tls/pq-support$ openssl ciphers -v 'ALL:COMPLEMENTOFALL' | grep TLSv1.3
TLS_AES_256_GCM_SHA384         TLSv1.3 Kx=any      Au=any   Enc=AESGCM(256)            Mac=AEAD
TLS_CHACHA20_POLY1305_SHA256   TLSv1.3 Kx=any      Au=any   Enc=CHACHA20/POLY1305(256) Mac=AEAD
TLS_AES_128_GCM_SHA256         TLSv1.3 Kx=any      Au=any   Enc=AESGCM(128)            Mac=AEAD
```

KEM Algorithms

```
ubuntu@strongswan:~/pqproto/tls/pq-support$ openssl list -kem-algorithms | grep -i mlkem
  { 2.16.840.1.101.3.4.4.1, id-alg-ml-kem-512, ML-KEM-512, MLKEM512 } @ default
  { 2.16.840.1.101.3.4.4.2, id-alg-ml-kem-768, ML-KEM-768, MLKEM768 } @ default
  { 2.16.840.1.101.3.4.4.3, id-alg-ml-kem-1024, ML-KEM-1024, MLKEM1024 } @ default
  X25519MLKEM768 @ default
  X448MLKEM1024 @ default
  SecP256r1MLKEM768 @ default
  SecP384r1MLKEM1024 @ default
```

Signature Algorithms

```
ubuntu@strongswan:~/pqproto/tls/pq-support$ openssl list -signature-algorithms | grep -i mldsa
  { 2.16.840.1.101.3.4.3.17, id-ml-dsa-44, ML-DSA-44, MLDSA44 } @ default
  { 2.16.840.1.101.3.4.3.18, id-ml-dsa-65, ML-DSA-65, MLDSA65 } @ default
  { 2.16.840.1.101.3.4.3.19, id-ml-dsa-87, ML-DSA-87, MLDSA87 } @ default
ubuntu@strongswan:~/pqproto/tls/pq-support$
```

```
ubuntu@strongswan:~/pqproto/tls/pq-support$ openssl list -signature-algorithms | grep -i slh
  { 2.16.840.1.101.3.4.3.20, id-slh-dsa-sha2-128s, SLH-DSA-SHA2-128s } @ default
  { 2.16.840.1.101.3.4.3.21, id-slh-dsa-sha2-128f, SLH-DSA-SHA2-128f } @ default
  { 2.16.840.1.101.3.4.3.22, id-slh-dsa-sha2-192s, SLH-DSA-SHA2-192s } @ default
  { 2.16.840.1.101.3.4.3.23, id-slh-dsa-sha2-192f, SLH-DSA-SHA2-192f } @ default
  { 2.16.840.1.101.3.4.3.24, id-slh-dsa-sha2-256s, SLH-DSA-SHA2-256s } @ default
  { 2.16.840.1.101.3.4.3.25, id-slh-dsa-sha2-256f, SLH-DSA-SHA2-256f } @ default
  { 2.16.840.1.101.3.4.3.26, id-slh-dsa-shake-128s, SLH-DSA-SHAKE-128s } @ default
  { 2.16.840.1.101.3.4.3.27, id-slh-dsa-shake-128f, SLH-DSA-SHAKE-128f } @ default
  { 2.16.840.1.101.3.4.3.28, id-slh-dsa-shake-192s, SLH-DSA-SHAKE-192s } @ default
  { 2.16.840.1.101.3.4.3.29, id-slh-dsa-shake-192f, SLH-DSA-SHAKE-192f } @ default
  { 2.16.840.1.101.3.4.3.30, id-slh-dsa-shake-256s, SLH-DSA-SHAKE-256s } @ default
  { 2.16.840.1.101.3.4.3.31, id-slh-dsa-shake-256f, SLH-DSA-SHAKE-256f } @ default
```

***Note**: More Algorithm and cipher suites can be used with liboqs/OQS Provider additions*

TLS Connection Phases

NGKORE

1. Setup Phase

- Create and configure SSL_CTX
- Configure cryptographic parameters:
  - *SSL_CTX_set1_sigalgs_list()* - signature algorithms
  - *SSL_CTX_set1_groups_list()* - supported groups
- Load certificates and private keys
- Create SSL object and associate with socket/BIO

2. Handshake Phase

- ClientHello/ServerHello exchange (cipher negotiation)
- Certificate verification and key exchange
- Complete with Finished messages

3. Application Data Transfer

- Secure bidirectional communication using established session keys
- Read/write application data through SSL object
- Protocol-level messages may still be exchanged alongside application data

4. Shutdown Phase

- Send *close_notify* alert to initiate graceful shutdown
- Receive *close_notify* response from peer
- Clean up SSL objects and close underlying sockets

TLS (PQ-Support): Terminal Logs

NGKORE

```
ubuntu@strongswan:~/pqproto/tls/pq-support$ ./server --port 8443
=== Post-Quantum TLS 1.3 Echo Server ===

SSL context created with TLS 1.3 enforcement (required for post-quantum)
Post-quantum signature algorithms configured
Post-quantum key exchange groups configured
Post-quantum server certificate loaded from ./certs/server-cert.pem
Post-quantum server private key loaded from ./certs/server-key.pem
Post-quantum certificate and private key match verified

=== Certificate & Key ===
Certificate algorithm: ML-DSA-65
Certificate key size: 15616 bits
Private key algorithm: ML-DSA-65 (15616 bits)
==========================

Post-quantum TLS server listening on port 8443
Post-quantum TLS 1.3 server ready. Press Ctrl+C to stop.
New connection from 127.0.0.1:35770
Starting post-quantum TLS handshake...
Post-quantum TLS handshake successful

=== TLS Connection ===
TLS version: TLSv1.3
Cipher suite: TLS_AES_256_GCM_SHA384
Key exchange: X25519MLKEM768
Server signature: mldsa65
======================

Client connected. Starting echo service...
Client sent: hi, there!
Server echoed: hi, there!
```

```
ubuntu@strongswan:~/pqproto/tls/pq-support$ ./client --host localhost --port 8443
=== Post-Quantum TLS 1.3 Client ===

SSL context created with TLS 1.3 enforcement (required for post-quantum)
Post-quantum signature algorithms configured for client
Post-quantum key exchange groups configured for client
Post-quantum CA certificate loaded from ./certs/ca-cert.pem
Post-quantum server certificate verification enabled
Resolved localhost to 127.0.0.1
Connected to post-quantum TLS server localhost:8443
SNI hostname set to: localhost
Hostname verification enabled for: localhost
Starting post-quantum TLS handshake...
Post-quantum TLS handshake successful!

=== TLS Connection ===
TLS version: TLSv1.3
Cipher suite: TLS_AES_256_GCM_SHA384
Key exchange: X25519MLKEM768
Peer signature algorithm: mldsa65
======================

=== Server Certificate ===
Certificate algorithm: ML-DSA-65
Certificate key size: 15616 bits
==================================

Post-quantum TLS connection established! Type messages to send to server (Ctrl+C to exit):
> hi, there!
Server echoed: hi, there!
```

← Echo Server

Client

mTLS (Mutual Transport Layer Security)

NGKORE

- **mTLS** connection setup is **identical to the standard TLS** client/server configuration
- The only difference is the **client certificate verification on the server side** -

SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER | SSL_VERIFY_FAIL_IF_NO_PEER_CERT, verify_client_callback)

- *pqproto Classical and PQ-Support mTLS1.3 Client/Server Setup* - *https://github.com/ngkore/pqproto/tree/main/mtls*

mTLS (PQ-support): Terminal Logs

NGKORE

```
ubuntu@strongswan:~/pqproto/mtls/pq-support$ ./server --port 8443
=== Post-Quantum mTLS 1.3 Echo Server ===

SSL context created with TLS 1.3 enforcement (required for post-quantum mTLS)
Post-quantum signature algorithms configured for mTLS server
Warning: Could not set post-quantum groups list
Server certificate loaded from ./certs/server-cert.pem
Server private key loaded from ./certs/server-key.pem
Server certificate and private key match verified
CA certificate loaded for client verification
Client certificate verification enabled (post-quantum mutual TLS)

=== Server Certificate & Key ===
Server certificate algorithm: ML-DSA-65
Server certificate key size: 15616 bits
Server private key algorithm: ML-DSA-65 (15616 bits)
================================

Post-quantum mTLS server listening on port 8443
Post-quantum mTLS 1.3 server ready. Press Ctrl+C to stop.
New connection from 127.0.0.1:37104
Starting post-quantum mTLS handshake...
Certificate verification: depth=1, preverify_ok=1
Certificate verification successful
Certificate verification: depth=0, preverify_ok=1
Certificate verification successful
Post-quantum mTLS handshake successful

=== Post-Quantum mTLS Connection ===
TLS version: TLSv1.3
Cipher suite: TLS_AES_256_GCM_SHA384
Key exchange: X25519MLKEM768
Server signature: mldsa65
Client signature: mldsa65
====================================

=== Client Certificate ===
Client subject: /C=US/ST=Test/L=Test/O=Test PQ Client/CN=Test PQ Client
Client issuer: /C=US/ST=Test/L=Test/O=Test PQ CA/CN=Test PQ CA
Client certificate algorithm: ML-DSA-65
Client certificate key size: 15616 bits
==========================

Client connected with post-quantum mTLS. Starting echo service...
Client sent: hi there, team!!
Server echoed: hi there, team!!
```

Echo Server

```
ubuntu@strongswan:~/pqproto/mtls/pq-support$ ./client --host localhost --port 8443
=== Post-Quantum mTLS 1.3 Client ===

SSL context created with TLS 1.3 enforcement (required for post-quantum mTLS)
Post-quantum signature algorithms configured for mTLS client
Warning: Could not set post-quantum groups list
Client certificate loaded from ./certs/client-cert.pem
Client private key loaded from ./certs/client-key.pem
Client certificate and private key match verified
CA certificate loaded from ./certs/ca-cert.pem
Server certificate verification enabled

=== Client Certificate & Key ===
Client subject: /C=US/ST=Test/L=Test/O=Test PQ Client/CN=Test PQ Client
Client certificate algorithm: ML-DSA-65
Client certificate key size: 15616 bits
Client private key algorithm: ML-DSA-65 (15616 bits)
================================

Resolved localhost to 127.0.0.1
Connected to post-quantum mTLS server localhost:8443
SNI hostname set to: localhost
Hostname verification enabled for: localhost
Starting post-quantum mTLS handshake...
Post-quantum mTLS handshake successful!

=== Post-Quantum mTLS Connection ===
TLS version: TLSv1.3
Cipher suite: TLS_AES_256_GCM_SHA384
Key exchange: X25519MLKEM768
Server signature algorithm: mldsa65
Client signature algorithm: mldsa65
====================================

=== Server Certificate ===
Server subject: /C=US/ST=Test/L=Test/O=Test PQ/CN=localhost
Server issuer: /C=US/ST=Test/L=Test/O=Test PQ CA/CN=Test PQ CA
Server certificate algorithm: ML-DSA-65
Server certificate key size: 15616 bits
==========================

Post-quantum mTLS connection established! Type messages to send to server (Ctrl+C to exit):
> hi there, team!!
Server echoed: hi there, team!!
```

Client

DTLS (Datagram Transport Layer Security)

NGKORE

- DTLS connection setup is slightly different from a standard TLS client/server configuration
- TLS runs over TCP, where reliability is critical, while **DTLS runs over UDP**, prioritizing speed and low latency
- Add **cookie secrets in the SSL context** for client verification to help mitigate DoS attacks.
- OpenSSL 3.5.3 does not yet support DTLS 1.3 (still in development):
  - Branch (dtls-1.3) - *https://github.com/openssl/openssl/tree/feature/dtls-1.3*
  - Available for testing with PQ-support using oqs-provider and liboqs
- Testing performed with **DTLS 1.2** client/server setup
- No post-quantum support in DTLS 1.2—implementation currently runs in **classical mode** only
- Prototype reference: *https://github.com/ngkore/pqproto/tree/main/dtls*

Comcast Innovation Fund supports work on DTLSv1.3

Oct 1, 2025

QUIC (Quick UDP Internet Connections)

- Traditional web stack uses TCP + TLS + HTTP: TCP ensures reliability, TLS provides encryption, and HTTP/2 offers efficiency
- **QUIC combines all three layers into one protocol**, streamlining connection setup and reducing latency.
- QUIC builds on top of UDP, not replacing TCP/UDP, but **avoiding protocol ossification** while maintaining flexibility
- OpenSSL QUIC introduction: *https://docs.openssl.org/master/man7/ossl-guide-quic-introduction/#what-is-quic*
- OpenSSL QUIC source: *https://github.com/openssl/openssl/tree/master/ssl/quic*
- Demo setups:
  - HTTP/1-based QUIC client/server - *https://github.com/openssl/openssl/tree/master/demos/quic*
  - HTTP/3 client/server - *https://github.com/openssl/openssl/tree/master/demos/http3*
- Single-stream QUIC client/echo server implemented in classical and post-quantum (PQ) support modes - *https://github.com/ngkore/pqproto/tree/main/quic*

QUIC: Key Functional Elements

NGKORE

- *SSL Context*
  - *SSL_CTX_new(_method_)* – create SSL context
  - *OSSL_QUIC_server_method() / OSSL_QUIC_client_method()* – server/client QUIC contexts
- Application Protocol Negotiation (ALPN)
  - *SSL_CTX_set_alpn_select_cb()* – mandatory during TLS handshake
  - Prototype used "http/1.0" instead of "h3" (HTTP/3)
- Use BIO sockets or create manual sockets
- **Single Streams:** *SSL_new_listener(SSL_CTX, flags) + SSL_accept_connection(SSL, flags)* for bidirectional stream
- **Multi Streams:** *SSL_new_stream() – see https://docs.openssl.org/master/man7/ossl-guide-quic-multi-stream/*
- Verify Handshake completion with *SSL_is_init_finished() as* PQ context increases handshake time

QUIC (PQ-support): Terminal Logs

NGKORE

```
ubuntu@strongswan:~/pqproto/quic/pq-support$ ./server --port 5433
=== Post-Quantum QUIC Echo Server ===

Post-quantum key exchange groups configured
ALPN protocols configured (demo: http/1.0, not actual HTTP/3)
QUIC SSL context created successfully
Server certificate loaded from ./certs/server-cert.pem
Server private key loaded from ./certs/server-key.pem
Certificate and private key match verified

=== Post-Quantum QUIC Server Certificate & Key ===
Server certificate algorithm: ML-DSA-65
Server certificate key size: 15616 bits
Server private key algorithm: ML-DSA-65 (15616 bits)
================================================

QUIC server bound to UDP port 5433
QUIC server ready on port 5433. Press Ctrl+C to stop.
Waiting for QUIC connection...
QUIC connection established
Handshake completed successfully

=== Post-Quantum QUIC Connection ===
QUIC version: QUICv1
Cipher suite: TLS_AES_256_GCM_SHA384
Key exchange: X25519MLKEM768
Server signature: mldsa65
====================================
QUIC client connected. Starting stream handling...
Stream 0 received 9 bytes: Hi there!
Echoed 9 bytes back to stream 0
QUIC client disconnected
QUIC connection handling completed
Shutting down QUIC connection...
QUIC connection closed
Waiting for QUIC connection...
^C
Received SIGINT, shutting down gracefully...
Failed to accept QUIC connection
QUIC connection closed
QUIC server shut down complete
```

← Echo Server

```
ubuntu@strongswan:~/pqproto/quic/pq-support$ ./client --host localhost --port 5433
=== Post-Quantum QUIC Client ===

Post-quantum key exchange groups configured for client
ALPN will be configured per-connection (demo: http/1.0)
QUIC SSL context created successfully
CA certificate loaded from ./certs/ca-cert.pem
Server certificate verification enabled
QUIC UDP socket created and connected
QUIC client configured for localhost:5433
Starting QUIC handshake...
QUIC handshake successful

=== QUIC Connection ===
QUIC version: QUICv1
Cipher suite: TLS_AES_256_GCM_SHA384
Key exchange: X25519MLKEM768
Peer signature algorithm: mldsa65
======================

=== Server Certificate ===
Server subject: /C=US/ST=CA/L=San Francisco/O=QUIC PQ Test Server/CN=localhost
Server issuer: /C=US/ST=CA/L=San Francisco/O=QUIC PQ Test CA/CN=QUIC PQ Test CA
Server certificate algorithm: ML-DSA-65
Server certificate key size: 15616 bits
============================

Using OpenSSL OpenSSL 3.5.3 16 Sep 2025 with QUIC support
QUIC connection established! Type messages to send to server (Ctrl+C to exit):
> Hi there!
Sent 10 bytes on QUIC stream
Received 9 bytes from QUIC stream
Server echoed: Hi there!
```

← Client

IPSec (Internet Protocol Security)

NGKORE

- Secures IP communication by encrypting data and authenticating sources; commonly used for VPNs
- Implemented a complete IPsec/IKEv2 VPN using StrongSwan v6.0.2 with OpenSSL 3.5.3 - *https://github.com/strongswan/strongswan*
- Built a Docker-based server–client network to run IPsec VPN in **tunnel mode** with configurable authentication: X.509 certificates or pre-shared keys
  - *https://github.com/ngkore/pqproto/tree/main/ipsec*
- Post-Quantum Setup:
  - Pre-shared keys: ML-KEM key exchange groups
  - Certificate auth: RSA certificates + ML-KEM key exchange
- StrongSwan (v6.0.2) currently does not support PQ certificates (e.g., ML-DSA) - *https://github.com/strongswan/strongswan/tree/ml-dsa* (development)

Strongswan based IPSec: Supported Algorithms

NGKORE

Selecting ML_KEM_768 for key exchange, Tunnel Mode

```
ubuntu@openssl:~$ docker exec ipsec-client swanctl --list-sas
connect-to-corporate: #27, ESTABLISHED, IKEv2, 44d298688713cbdc_i* f26a1083039f63ce_r
  local  'C=US, ST=Test, L=Test, O=Test Client, OU=Test, CN=client.test.local' @ 172.21.0.20[4500]
  remote 'C=US, ST=Test, L=Test, O=Test Server, OU=Test, CN=vpn.test.local' @ 172.21.0.10[4500]
  AES_GCM_16-256/PRF_HMAC_SHA2_384/ML_KEM_768
  established 493s ago, rekeying in 2791s, reauth in 9975s
  tunnel-to-corp: #15, reqid 1, INSTALLED, TUNNEL, ESP:AES_GCM_16-256/ESN
    installed 493s ago, rekeying in 2931s, expires in 3467s
    in  c5ae11eb,    263 bytes,     4 packets,     3s ago
    out c7712d09,    344 bytes,     6 packets,   185s ago
    local  172.21.0.20/32
    remote 10.1.0.1/32
```

A testing Python script to set up an interactive session between client and server via IPSec tunnel -

```
ubuntu@openssl:~$ docker exec -it ipsec-client python3 /usr/local/bin/test-client.py
==================================================
IPsec VPN Test Client
==================================================
[CLIENT] Connecting to 10.1.0.1:8080
[CLIENT] Connection will be encrypted via IPsec tunnel
[CLIENT] Connected successfully!
[CLIENT] Type messages to send (or 'quit' to exit)

==================================================

[CLIENT] Enter message: Hi there, OpenSSL Team!!
[CLIENT] [14:26:46] SENDING: 'Hi there, OpenSSL Team!!'
[CLIENT] [14:26:46] RECEIVED: 'SERVER_ECHO[14:26:46]: Hi there, OpenSSL Team!!'

[CLIENT] Enter message:
```

ML-KEM under Key Exchange Groups

```
ke {
  MODP_3072 = openssl
  MODP_4096 = openssl
  MODP_6144 = openssl
  MODP_8192 = openssl
  MODP_2048 = openssl
  MODP_2048_224 = openssl
  MODP_2048_256 = openssl
  MODP_1536 = openssl
  MODP_1024 = openssl
  MODP_1024_160 = openssl
  MODP_768 = openssl
  MODP_CUSTOM = openssl
  ML_KEM_512 = openssl
  ML_KEM_768 = openssl
  ML_KEM_1024 = openssl
  ECP_256 = openssl
  ECP_384 = openssl
  ECP_521 = openssl
  ECP_224 = openssl
  ECP_192 = openssl
  ECP_256_BP = openssl
  ECP_384_BP = openssl
  ECP_512_BP = openssl
  ECP_224_BP = openssl
  CURVE_25519 = openssl
  CURVE_448 = openssl
}
```

OAuth2.0 (ongoing)

- An authorization framework that lets users grant third-party apps limited access to protected resources without sharing passwords
- Using Ory Hydra – an open-source OAuth 2.0 & OpenID Connect (OIDC) authorization server in Go
  - *https://github.com/ory/hydra*
- Built-in JWT access token support
- Post-Quantum Integration:
  - Custom JWT signing through Go crypto interfaces using:
    - Cloudflare go or Circl (or standard Go crypto with PQ support) - *https://github.com/cloudflare/circl*
    - liboqs-go for ML-DSA signatures - *https://github.com/open-quantum-safe/liboqs-go*

SSH (ongoing)

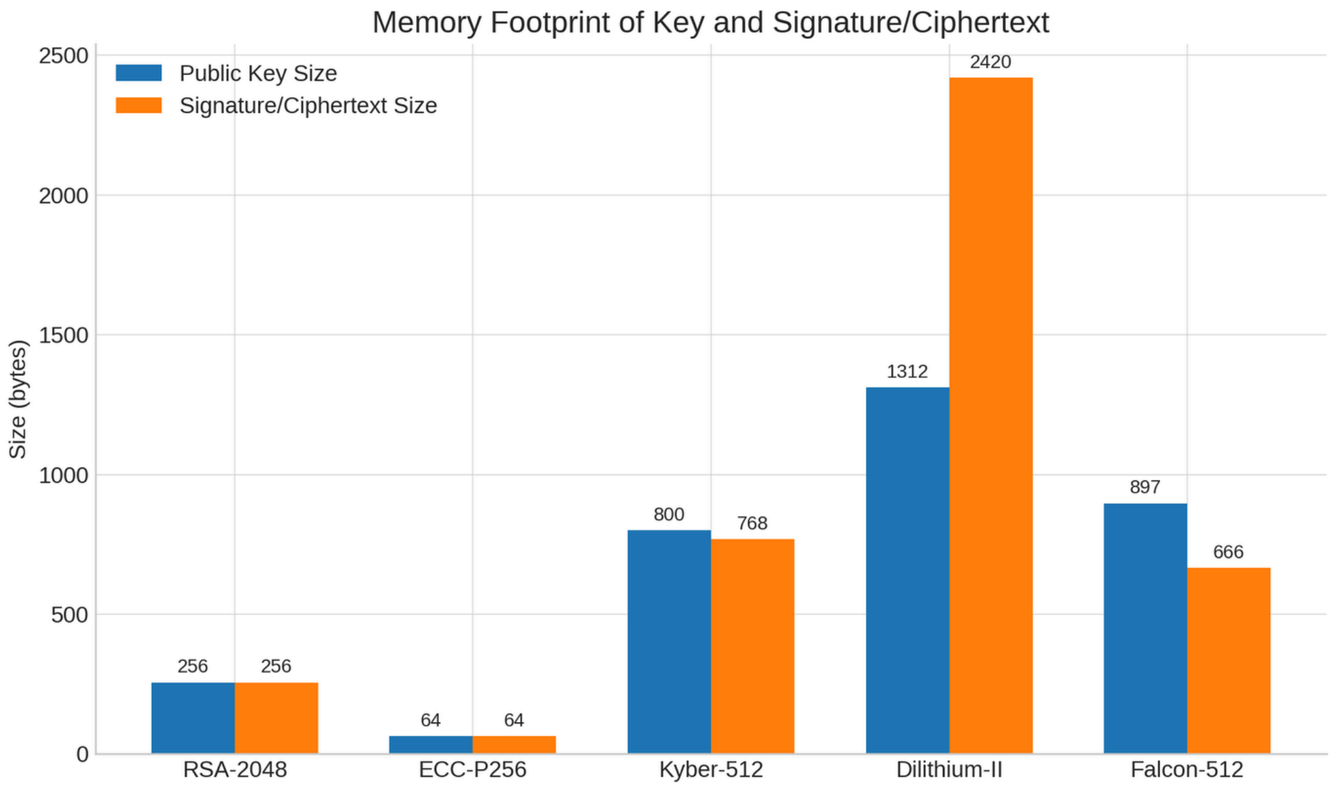
NGKORE

- A secure network protocol for encrypted remote communication and data transfer between a client and a server
- Fork of OpenSSH that includes prototype quantum-resistant key exchange and authentication in SSH based on liboqs - *https://github.com/open-quantum-safe/openssh*
- Build a server–client prototype setup using Go crypto/ssh/
- Integrating Cloudflare CIRCL library for post-quantum key exchange and signatures - *https://github.com/cloudflare/circl*

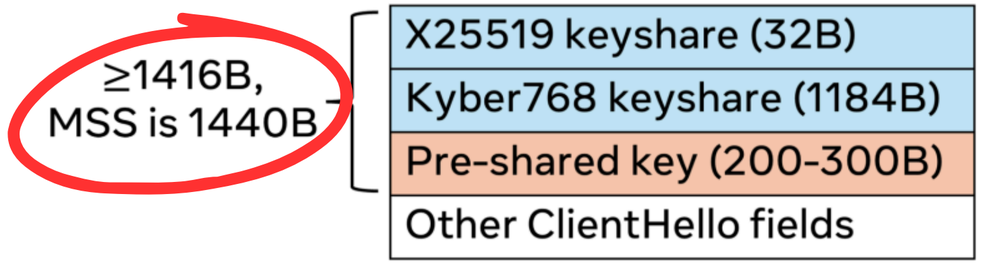
Challenges for PQC Migration

Challenges for PQC Migration (I)

- Key & Signature Size Explosion
  - ML-DSA-65 signatures: 3,309 bytes vs 64 bytes (Ed25519) → **certificate bloat and larger handshakes**
  - ML-KEM-768 keys: 1,184 bytes vs 32 bytes (X25519) → **~37× increase**



ClientHello size, when including ECDHE keyshares and PSK, will exceed MSS

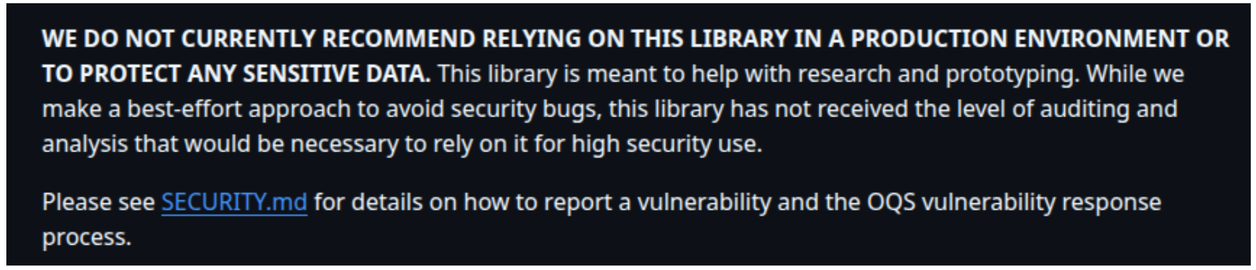


***Source**: https://engineering.fb.com/2024/05/22/security/post-quantum-readiness-tls-pqr-meta/*

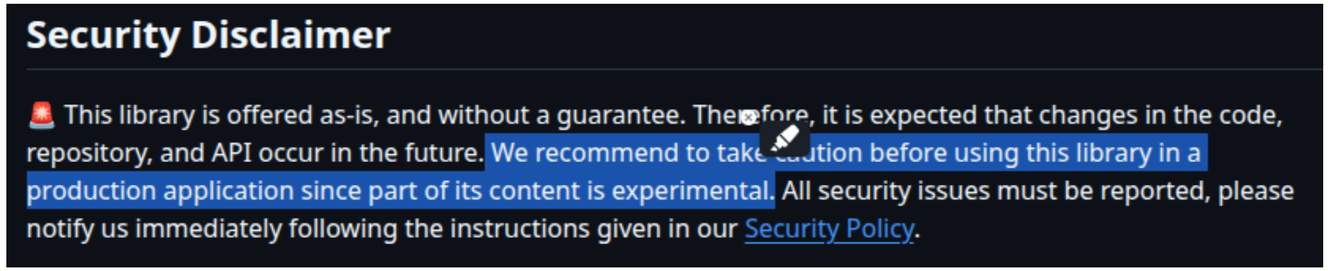
***Source**: https://arxiv.org/abs/2508.00832*

Challenges for PQC Migration (II)

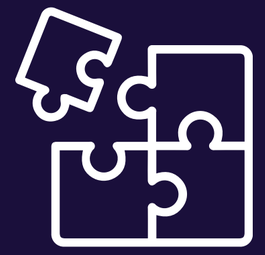
- **Lack of** full PQC migration (**native support**), **requiring fallback mechanisms** and hybrid deployments - **adds** configuration and maintenance **complexity**
- **Side-channel exposures**: timing attacks (e.g., *KyberSlash*, *Clangover*), fault-injection risks
- **Immature open-source PQC libraries** are not yet hardened for production
- static **buffer** allocations (e.g., 256-byte RSA) may **overflow with larger PQ keys**
- Network impact: TCP congestion window limits, DTLS fragmentation from oversized packets



Source: *https://github.com/open-quantum-safe/liboqs*



Source: *https://github.com/cloudflare/circl*



Migration Strategy & Best Practices

Migration Strategy & Best Practices

NGKORE

- **Identify high-risk systems vulnerable to HNDL attacks** and **plan their migration first**
- **Deploy hybrid** key exchanges combining classical and PQC algorithms **to maintain backward compatibility** during transition
- **Upgrade CA software to issue PQC or hybrid certificates** and handle larger key sizes
- **Start with pilot testing** (sooner) and gradually extend PQC support to production environments
- **Validate interoperability and performance across multiple implementations** such as OpenSSL, BoringSSL, wolfSSL, strongSwan, Circl, etc
- **Perform** constant-time code **audits**, fault injection, and fuzzing **to detect timing or side-channel vulnerabilities unique to PQC**
- **Measure handshake latency, throughput, and resource overhead** before and after PQC integration; adjust buffer sizes, caching, and session resumption for efficiency
- **Use hardware and algorithm optimizations**—AES-NI, SIMD, ADX, and NTT—to improve cryptographic performance

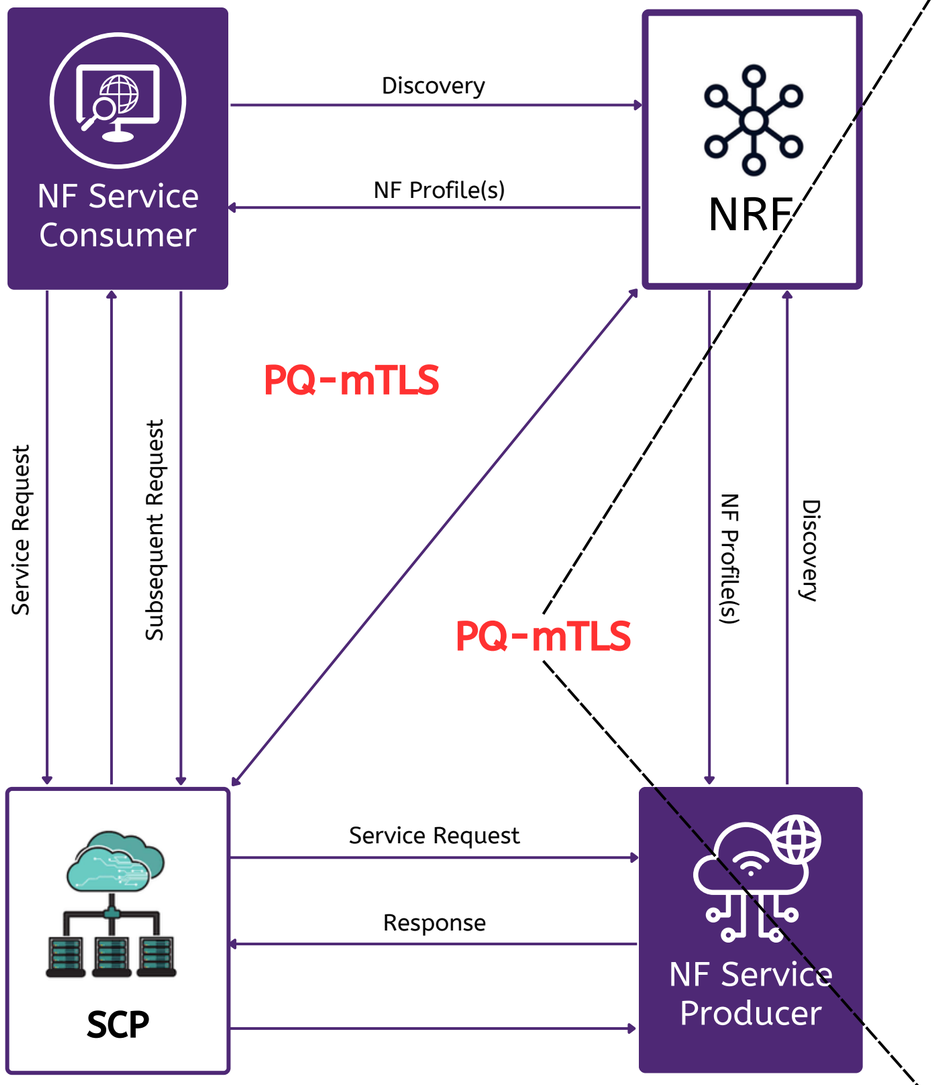
Real World Implementations

Our Implementations

- **PQ-DTLS1.3** (branch *feature/dtls-1.3*) over **F1AP interface** EURECOM **OpenAirInterface (OAI) RAN**
- **PQ-mTLS1.3** over Service Based Interface (**SBI**) in **Aether/SD-Core**
- **PQ-IPSec** (PSK-based Authentication) over **N3 interface** between **OAI RAN and OAI UPF**
- **PQ-OAuth2.0 Authorization** between **NRF and NFs in Free5GC**
- Implemented **PQC in 5G SUPI Concealment** for **Forward Secrecy in Open5GS**

** PQ-"protocol" refers "protocol" with PQ-support*

Use Case: PQ-mTLS in SBI in 5G Core Network



Field	Value
CA Type	Private (Internal) CA
Certificate Signature Algorithm	Homogeneous: ML-DSA-44/65/87
	Hybrid: ML-DSA_Ed448, ML-DSA_Ed25519 + Any TLS v1.3/1.2 ClassicalSignature Schemes
Signature Length	3293 octets + Classical Signature length (e.g. 64 octets for Ed25519)
Key Exchange Mechanism	Homogeneous: ML-KEM (512/768/1024)
	Hybrid: ECDHE_ML-KEM (e.g. X25519MLKEM768)
Key Exchange Length (Public key)	1216 octets (ML-KEM_768: 1184 octets, X25519/P256: 32 octets)
Key Exchange CipherText Length	1088 octets
AEAD - Symmetric Encryption & Authentication Algorithm	AES256_GCM, ChaCha20_Poly1305
enckeylen	32 octets (256 bits)
ivlen	12 octets (96 bits) (AEAD specific)
mackeylen	32/48 octets (256/384 bits)
maclen	32/48 octets (256/384 bits)
TLS KDF	HMAC-based Expand & Extract KDF (HKDF) / PRF for TLS v1.2
TLS Finished MAC algorithm	HMAC-SHA-256/384
Fallback Methods	TLS v1.2, Classical signature algorithms & Classical key exchanges supported

Example: PQ-mTLS in SBI in Aether/SD-Core

NGKORE

1.

```
New PQ TLS connection established.
------------------------------
Connection Details:

        1. Client Connected to: nrf
        2. Client Supported Signature Schemes:          PSSWithSHA256           ECDSAWithP256AndSHA256          Ed25519         PSSWithSHA384           PSSW
ithSHA512               PKCS1WithSHA256         PKCS1WithSHA384         PKCS1WithSHA512         ECDSAWithP384AndSHA384          ECDSAWithP521AndSHA512      P
KCS1WithSHA1            ECDSAWithSHA1           SignatureScheme(65121)          SignatureScheme(65122)
        3. Client Supported Curves:
                •X25519-Kyber768-Draft00
                •P256-Kyber768-Draft00
                •X25519
                •P-256
                •P-384
                •P-521

        4. Curve Preferences:
                •X25519-Kyber768-Draft00
                •X25519
                •P-256
SERVER Certificate found.
```

ML-DSA Certs

Hybrid Curve Pref.

Client (NF) Supported Signature Schemes, Curves Preferences

RAN detection and connection establishment by 5G Core

3.

```
================================================
################|GNB DETECTED|#################
================================================
SCTP Accept from: | 192.168.130.80:54341 |
Create a new NG connection for: | 192.168.130.80:54341 |
Handle NG Setup request
Supported Tai List in HEXA-AMF Plmn: &{001 01}, Tac: 0x000001 Tac: 1
Supported Tai List in HEXA-AMF Plmn: &{001 01}, Tac: 0x000002 Tac: 2
Send NG-Setup response
Handle SCTP Notification[addr: 192.168.130.80:54341]
SCTP_SHUTDOWN_EVENT notification, close the connection
Remove RAN Context[ID: <PlmnID: {Mcc:001 Mnc:01}, GNbID: 0000e00>]
Handle SCTP Notification[addr: <nil>]
RAN context has been removed[addr: <nil>]
```

UE (Mobile) connected with 5G Core and a default PDU session is established b/w UE and Core Network

4.

```
================================================
################|PDU CREATED|#################
================================================
Send PDU Session Resource Setup Request
+-------------------+-----------------------------+
|AMF_UE_NGAP_ID     |7682023                      |
|IMSI               |imsi-001010100007494         |
|MCC                |001                          |
|MNC                |01                           |
|TAC                |000001                       |
+-------------------+-----------------------------+
```

NF (e.g., UDR, NSSF, PCF) registration with NRF and NF Profile creation

2.

```
Handle NFRegisterRequest
HearBeat Timer value: %v sec 86400
urilist create
urilist create
urilist create
Create NF Profile  UDR
Create NF Profile  NSSF
Create NF Profile  PCF
Location header:  https://nrf:29510/nnrf-nfm/v1/nf-instances/5d9d27df-22c9-4d37-bd42-15d3044cb260
Location header:  https://nrf:29510/nnrf-nfm/v1/nf-instances/0953d0d7-d384-459b-adee-f922f0b22c07
Location header:  https://nrf:29510/nnrf-nfm/v1/nf-instances/f387ce53-634f-457f-9572-d0df3493d874
| 201 |   192.168.84.49 | PUT     | /nnrf-nfm/v1/nf-instances/0953d0d7-d384-459b-adee-f922f0b22c07 |
| 201 |   192.168.84.33 | PUT     | /nnrf-nfm/v1/nf-instances/5d9d27df-22c9-4d37-bd42-15d3044cb260 |
| 201 |   192.168.84.17 | PUT     | /nnrf-nfm/v1/nf-instances/f387ce53-634f-457f-9572-d0df3493d874 |
```

Real World Applications (External Users)

NGKORE

Abstract

Enterprise cloud ecosystems face unprecedented security challenges as quantum computing threatens traditional cryptographic foundations underpinning federated API communications. This work presents a novel quantum-resilient middleware layer (QRIL) that integrates NIST-standardized post-quantum cryptographic algorithms, specifically CRYSTALS-Kyber for key encapsulation and Falcon for digital signatures, into enterprise API security frameworks. The middleware architecture employs hybrid encryption mechanisms, maintaining backward compatibility through combined ECC and post-quantum protocols, while enabling seamless federated identity propagation across multi-domain cloud platforms. Core components include quantum-safe proxy gateways, security kernel modules for policy-based access controls, and interoperability bridges supporting legacy system integration. Implementation leverages industry-standard libraries, including BouncyCastle, OpenQuantumSafe, and liboqs, integrated with enterprise Key Management Services across AWS, Azure, and IBM cloud platforms. Quantum attack simulations using IBM Qiskit and Microsoft Quantum Development Kit validate resistance against Grover's and Shor's algorithms. Performance evaluations demonstrate acceptable latency overhead while maintaining high throughput for quantum-safe handshake processes. The framework successfully addresses backward compatibility requirements while providing comprehensive protection against quantum cryptanalysis threats in enterprise federated API environments, establishing a foundation for quantum-safe digital transformation in critical infrastructure sectors.

Source: *https://lorojournals.com/index.php/emsj/article/view/1506*

android / platform / external / **liboqs**

Bug: 337064740

Clone this repo:

```
git clone https://android.googlesource.com/platform/external/liboqs
```

Source: *https://android.googlesource.com/platform/external/liboqs/*

How Meta is enabling PQC

Meta's TLS protocol library, Fizz, is designed for high security, reliability, and performance. The early work on Fizz previously helped standardize TLS 1.3 (RFC 8446). Fizz now supports a range of features including various handshake modes, PSK resumption, Diffie-Hellman key exchange authenticated with a pre-shared key for forward secrecy, async I/O, zero copy encryption, client authentication, and HelloRetryRequest. The use of our own implementation has allowed us to quickly react to new features in the TLS protocol.

Fizz is mostly built on top of three libraries: Folly, OpenSSL, and Sodium. To support PQC, we make use of liboqs, which is an open source library led by world-renowned PQC experts that has received attention from both academia and industry experts. The liboqs library implements post-quantum cryptography algorithms for key encapsulation and signature mechanisms, including Kyber. Additionally, we extended Fizz with hybrid key exchange functionality, which can make use of the new post-quantum key exchange mechanisms provided by liboqs alongside existing classical mechanisms.

Source: *https://engineering.fb.com/2024/05/22/security/post-quantum-readiness-tls-pqr-meta/*

External users of OQS

liboqs has been used in the following external projects:

- Meta:
  - Post-quantum readiness for TLS at Meta
  - Meta is getting ready for post-quantum cryptography (podcast, liboqs mentioned at the 25-minute mark)
- Utimaco Hardware Security Module demo with evolutionQ
- Microsoft Post-Quantum Cryptography VPN: Experimental fork of OpenVPN adding post-quantum cryptography, to evaluate functionality and performance of quantum resistant cryptography in a VPN setting.
- Mullvad VPN: Public beta of Mullvad's VPN client using post-quantum key exchange.
- Thales eSecurity Go wrapper
- Liesware Coherence Cryptographic Server
- Senetas/Thales High Speed Encryption: implementation of hybrid RSA/EC and QRA/QKD
- Cisco: Post-Quantum TLS 1.3 and SSH Performance (preliminary results)
- strongSwan: Post-Quantum IKEv2 demo
- IBM Cloud
- mbedTLS

Source: *https://openquantumsafe.org/applications/external.html*

Introducing Post-Quantum Cryptography for IPsec in IPFire

by Michael Tremer, March 26

With the upcoming release of IPFire 2.29 Core Update 193, we are excited to announce the integration of post-quantum cryptography (PQC) for IPsec, thanks to the recent release of strongSwan 6.0.0. This marks a significant step forward in securing communications against future threats posed by quantum computing.

Source: *https://www.ipfire.org/blog/introducing-post-quantum-cryptography-for-ipsec-in-ipfire*

RHEL 10 packages `liboqs`, `oqsprovider`, `nss`, `openssh`, and `gnutls` provide PQC as a Technology Preview

The RHEL 10.0 packages `liboqs`, `oqsprovider`, `nss`, `openssh`, and `gnutls` provide post-quantum cryptography (PQC) as a Technology Preview. To enable the PQC algorithms, install the `crypto-policies-pq-preview` package and apply the `TEST-PQ` cryptographic subpolicy.

Source: *https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/10/html/10.0_release_notes/technology-preview-features*

Thank You

Q&A

OpenSSL Conference Prague 2025