RFC 10024 : Post-Quantum Traditional (PQ/T) Hybrid Key Agreement Mechanisms for TLS 1.3

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 20 août 2026

Date de publication du RFC : Août 2026

https://www.bortzmeyer.org/10024.html

On met du post-quantique partout en ce moment. Ce court RFC normalise l'utilisation de clés hybrides (post-quantiques et traditionnelles) dans les échanges de clés de TLS.

Les trois mécanismes normalisés dans ce RFC utilisent tous ML-KEM, qui est normalisé dans FIPS-203 <https://csrc.nist.gov/pubs/fips/203/final>. ML-KEM repose sur les réseaux euclidiens et les experts en cryptographie ne sont pas sûrs à 100 % de sa sécurité. Dans le doute, il est donc recommandé de l'utiliser combiné avec un algorithme traditionnel, comme décrit dans le RFC 9954[1]. Ainsi, on est protégé à la fois contre les futurs CRQC ("*Cryptographically-Relevant Quantum Computer*") et contre la cryptanalyse plus classique. Les principes de cette méthode hybride sont décrits en détail dans le RFC 9794 et le RFC 9954 déjà cité.

Les trois mécanismes d'échange de clés <https://www.bortzmeyer.org/echange-cles.html> pour TLS normalisés ici (et qui sont appelés « groupes » pour des raisons historiques) sont :

— X25519MLKEM768, qui combine l'algorithme traditionnel à courbes elliptiques X25519 (RFC 7748) avec ML-KEM, et qui est le mécanisme recommandé.
— SecP256r1MLKEM768, qui combine SecP256 (norme FIPS-186 <https://csrc.nist.gov/pubs/fips/186-5/final>, donc une courbe elliptique différente, la P256) avec ML-KEM, et qui a l'avantage ou l'inconvénient de n'utiliser que des algorithmes NIST.
— SecP384MLKEM1024, proche du précédent mais avec des clés plus longues et une courbe elliptique différente, avec normalement davantage de sécurité.

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc9954.txt

Pour les trois mécanismes traditionnels, avec des courbes elliptiques, l'algorithme ECDH est utilisé.

Pour réaliser l'échange de clés au début d'une session TLS, on fait tourner le mécanisme traditionnel **et** le post-quantique, et on concatène les deux clés. (Oui, je simplifie, que les experts en cryptographie m'excusent, la sortie d'ECDH utilisée n'est pas vraiment la clé.) On notera que, pour X25519MLKEM768, la clé faite via ML-KEM fait 1 184 octets et celle d'ECDH seulement 32 octets ; c'est un problème avec beaucoup d'algorithmes post-quantiques, dont les clés et les signatures sont très longues. (Et, oui, on n'utilise pas directement ces « clés » mais un dérivé.)

Si on se soucie de conformité plutôt que de sécurité, on lira avec bonheur la section 5 du RFC, qui décrit en détail la conformité de ce RFC avec les règles FIPS (c'est surtout utile si vous travaillez pour le gouvernement étatsunien).

Et si vous vous intéressez aux débats sur la sécurité de ML-KEM et sur le concept même d'hybride, parmi les nombreux articles publiés, je vous recommande celui de Sophie Schmieg <https://keymaterial.net/2025/11/27/ml-kem-mythbusting/>.

Les trois mécanismes d'échange de clés ont été ajoutés au registre IANA <https://www.iana.org/assignments/tls-parameters/tls-parameters.xml#tls-parameters-8>, avec respectivement les numéros 4588, 4587 et 4589.

Voici, vus par Firefox (« Outils de développement Web » puis « Réseau » puis « Sécurité », tout au bout), les algorithmes cryptographiques utilisés avec un serveur HTTPS qui accepte un des mécanismes de notre RFC. Il y a en tout trois éléments à la liste car il faut trois choses pour faire du TLS :

— Un algorithme d'échange de clés (c'est le sujet de ce RFC), ici mlkem768x25519 (X25519MLKEM768 dans le RFC, c'est aussi ce qu'afficherait Chrome),
— Un algorithme d'authentification via une signature, ici RSA-PSS-SHA256 (c'est l'algorithme que vous trouverez dans le certificat),
— Un algorithme pour le chiffrement symétrique, une fois que les clés de session auront été échangées, ici ChaCha20 avec Poly1305.

Seul le premier des trois cités inclut du post-quantique. RSA pourrait faire l'échange de clés (une des parties génère une clé de session et la chiffre avec RSA avant de l'envoyer) et la signature mais ML-KEM ne sait faire que l'échange de clés.

À part Firefox, Chrome gère ces nouveaux mécanismes (qui sont déjà dans plusieurs bibliothèques de cryptographie, cf. par exemple la liste de Cloudflare <https://developers.cloudflare.com/ssl/post-quantum-cryptography/pqc-support/>), ainsi que des hébergeurs comme Cloudflare ou AWS. Voici une connexion vue par Chrome :

Jouons un peu avec OpenSSL maintenant. On va utiliser la version qui est dans Debian stable. D'abord, créons un certificat pour ECDSA :

```
% openssl version
OpenSSL 3.5.4 30 Sep 2025 (Library: OpenSSL 3.5.4 30 Sep 2025)

% openssl ecparam -out server.pem -name prime256v1 -genkey

% openssl req -new -key server.pem  -nodes -out server.csr
...
Country Name (2 letter code) [AU]:FR
...

% openssl x509 -in server.csr -out server.crt -req -signkey server.pem -days 2001
Certificate request self-signature ok
subject=C=FR...
```

Maintenant qu'on a le certificat, lançons le serveur en lui disant qu'on accepte d'échanger les clés en X25519MLKEM768 (et qu'on écoute sur le port 12345) :

```
% openssl s_server -tls1_3 -groups X25519MLKEM768 -accept 12345 -cert server.crt -key server.pem
```

Firefox va pouvoir s'y connecter et affichera :

```
Version du protocole : "TLSv1.3"
Suite de chiffrement : "TLS_AES_128_GCM_SHA256"
Groupe d'échange de clés : "mlkem768x25519"
Algorithme de signature : "ECDSA-P256-SHA256"
```

Le serveur OpenSSL annonce quant à lui :

```
Shared ciphers:TLS_AES_128-GCM-SHA256:ECDHE-ECDSA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305:ECDHE-ECDSA-AES2
Signature Algorithms: ECDSA+SHA256:ECDSA+SHA384:ECDSA+SHA512:RSA-PSS+SHA256:RSA-PSS+SHA384:RSA-PSS+SHA512:RSA+SH
Shared Signature Algorithms: ECDSA+SHA256:ECDSA+SHA384:ECDSA+SHA512:RSA-PSS+SHA256:RSA-PSS+SHA384:RSA-PSS+SHA512
Supported groups: X25519MLKEM768:x25519:secp256r1:secp384r1:secp521r1:ffdhe2048:ffdhe3072
Shared groups: X25519MLKEM768
CIPHER is TLS_AES_128_GCM_SHA256
```

À noter que si on avait lancé le serveur avec `-groups MLKEM768`, Firefox ne pouvait se connecter (*"no suitable key share"*) puisqu'il ne gère pas ML-KEM seul, uniquement dans un contexte hybride (ML-KEM et ECDH-X25519). L'usage de ML-KEM seul est décrit dans l'*"Internet draft"* `draft-ietf-tls-mlkem`. Et si vous voulez connaitre tous les mécanismes d'échange de clés de votre installation d'OpenSSL :

```
% openssl list -kem-algorithms
  { 1.2.840.113549.1.1.1, 2.5.8.1.1, RSA, rsaEncryption } @ default
  { 1.2.840.10045.2.1, EC, id-ecPublicKey } @ default
  { 1.3.101.110, X25519 } @ default
  { 1.3.101.111, X448 } @ default
  { 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
```

(Pour ceux de signature, c'est `openssl list -signature-algorithms` et, logiquement, on n'y trouve pas ML-KEM.)

Et dans curl? Je n'ai pas testé mais normalement ceci marche <https://github.com/curl/everything-curl/blob/master/usingcurl/tls/post-quantum.md> :

```
curl -v --curves X25519MLKEM768 https://example.com
```